PB80-168545

Using ANS FORTRAN

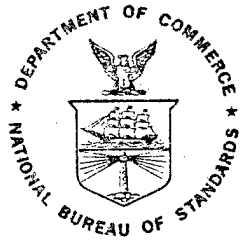
(U.S.) National Bureau of Standards
Washington, DC

Prepared for

National Science Foundation
Washington, DC

Mar 80

U.S. DEPARTMENT OF COMMERCE
National Technical Information Service
NTIS®



NBS HANDBOOK 131

U.S. DEPARTMENT OF COMMERCE / National Bureau of Standards

Using ANS FORTRAN

NATIONAL BUREAU OF STANDARDS

The National Bureau of Standards[1] was established by an act of Congress on March 3, 1901. The Bureau's overall goal is to strengthen and advance the Nation's science and technology and facilitate their effective application for public benefit. To this end, the Bureau conducts research and provides: (1) a basis for the Nation's physical measurement system, (2) scientific and technological services for industry and government, (3) a technical basis for equity in trade, and (4) technical services to promote public safety. The Bureau's technical work is performed by the National Measurement Laboratory, the National Engineering Laboratory, and the Institute for Computer Sciences and Technology.

THE NATIONAL MEASUREMENT LABORATORY provides the national system of physical and chemical and materials measurement; coordinates the system with measurement systems of other nations and furnishes essential services leading to accurate and uniform physical and chemical measurement throughout the Nation's scientific community, industry, and commerce; conducts materials research leading to improved methods of measurement, standards, and data on the properties of materials needed by industry, commerce, educational institutions, and Government; provides advisory and research services to other Government agencies; develops, produces, and distributes Standard Reference Materials; and provides calibration services. The Laboratory consists of the following centers:

Absolute Physical Quantities[2] — Radiation Research — Thermodynamics and Molecular Science — Analytical Chemistry — Materials Science.

THE NATIONAL ENGINEERING LABORATORY provides technology and technical services to the public and private sectors to address national needs and to solve national problems; conducts research in engineering and applied science in support of these efforts; builds and maintains competence in the necessary disciplines required to carry out this research and technical service; develops engineering data and measurement capabilities; provides engineering measurement traceability services; develops test methods and proposes engineering standards and code changes; develops and proposes new engineering practices; and develops and improves mechanisms to transfer results of its research to the ultimate user. The Laboratory consists of the following centers:

Applied Mathematics — Electronics and Electrical Engineering[2] — Mechanical Engineering and Process Technology[2] — Building Technology — Fire Research — Consumer Product Technology — Field Methods.

THE INSTITUTE FOR COMPUTER SCIENCES AND TECHNOLOGY conducts research and provides scientific and technical services to aid Federal agencies in the selection, acquisition, application, and use of computer technology to improve effectiveness and economy in Government operations in accordance with Public Law 89-306 (40 U.S.C. 759), relevant Executive Orders, and other directives; carries out this mission by managing the Federal Information Processing Standards Program, developing Federal ADP standards guidelines, and managing Federal participation in ADP voluntary standardization activities; provides scientific and technological advisory services and assistance to Federal agencies; and provides the technical foundation for computer-related policies of the Federal Government. The Institute consists of the following centers:

Programming Science and Technology — Computer Systems Engineering.

[1]Headquarters and Laboratories at Gaithersburg, MD, unless otherwise noted; mailing address Washington, DC 20234.
[2]Some divisions within the center are located at Boulder, CO 80303.

NBS-114A (REV. 9-78)

U.S. DEPT. OF COMM. BIBLIOGRAPHIC DATA SHEET	1. PUBLICATION OR REPORT NO. NBS Handbook 131	2. Gov't. Accession No.	3. Recipient's Accession No. PB80-168545

4. TITLE AND SUBTITLE

Using ANS FORTRAN

5. Publication Date

March 1980

6. Performing Organization Code

7. AUTHOR(S)

G. E. Lyon, Editor

8. Performing Organ. Report No.

9. PERFORMING ORGANIZATION NAME AND ADDRESS

NATIONAL BUREAU OF STANDARDS
DEPARTMENT OF COMMERCE
WASHINGTON, DC 20234

10. Project/Task/Work Unit No.

11. Contract/Grant No.

(NSF) DCR 75-045-443

12. SPONSORING ORGANIZATION NAME AND COMPLETE ADDRESS *(Street, City, State, ZIP)*

Same as 9 above and
National Science Foundation
1900 Pennsylvania Avenue, NW
Washington, DC 20550

13. Type of Report & Period Covered

Final

14. Sponsoring Agency Code

15. SUPPLEMENTARY NOTES

Library of Congress Catalog Card Number: 80-600009

☐ Document describes a computer program; SF-185, FIPS Software Summary, is attached.

16. ABSTRACT *(A 200-word or less factual summary of most significant information. If document includes a significant bibliography or literature survey, mention it here.)*

This FORTRAN volume presents, in order: a set of quick and clear reference charts for ANS FORTRAN 66 syntax; observations on using only standard FORTRAN 66 features; instructions on circumventing and extending FORTRAN 66 with the least harm; an appraisal of the new FORTRAN 77 in terms of FORTRAN 66 constructs. Although the chapters comprise much material that has appeared in other technical memoranda or published articles, heavily recast sections have been re-refereed.

The four chapters address programmers concerned with FORTRAN transportability, managers engaged in programming standards, and other practitioners interested in system influences upon languages. Since the text touches upon several general programming aspects (input/output, storage allocation, storage lifetimes and protection, control structures), the volume's appeal will extend beyond the immediate FORTRAN community.

17. KEY WORDS *(six to twelve entries; alphabetical order; capitalize only the first letter of the first key word unless a proper name; separated by semicolons)*

ANS FORTRAN; FORTRAN 77; standard programming language; transferability.

18. AVAILABILITY ☒ Unlimited

☐ For Official Distribution. Do Not Release to NTIS

☐ Order From Sup. of Doc., U.S. Government Printing Office, Washington, DC 20402, SD Stock No. SN003-003-02165-2

☒ Order From National Technical Information Service (NTIS), Springfield, VA. 22161

19. SECURITY CLASS (THIS REPORT)	21. NO. OF PRINTED PAGES
UNCLASSIFIED	
20. SECURITY CLASS (THIS PAGE)	**22. Price**
UNCLASSIFIED	

USCOMM-DC

Using ANS FORTRAN

Gordon Lyon, Editor

Institute for Computer Sciences and Technology
National Bureau of Standards
Washington, D.C. 20234

Contributors:

Frances E. Holberton

Institute for Computer Sciences and Technology
National Bureau of Stantards
Washington, D.C. 20234

J. Larmouth

University of Salford
Salford, England

M. D. McIlroy

Bell Telephone Laboratories, Incorporated
Murray Hill, N.J. 07974

Sponsored by:
National Science Foundation
1900 Pennsylvania Avenue, NW
Washington, D.C. 20550

U.S. DEPARTMENT OF COMMERCE, Philip M. Klutznick, Secretary

Luther H. Hodges, Jr., Deputy Secretary

Jordan J. Baruch, Assistant Secretary for Science and Technology

NATIONAL BUREAU OF STANDARDS, Ernest Ambler, Director

Issued March 1980

i.a

Library of Congress Catalog Card Number: 80-600009

National Bureau of Standards Handbook 131
Nat. Bur. Stand. (U.S.), Handb. 131, 106 pages (Mar. 1980)
CODEN: NBSHAP

U.S. GOVERNMENT PRINTING OFFICE
WASHINGTON: 1980

Foreword

Project background. Using ANS FORTRAN falls into a software engineering series first proposed by S. Jeffery of the National Bureau of Standards. Original handbook plans were formulated by a planning committee sponsored by the National Bureau of Standards, The National Science Foundation and the Association for Computing Machinery [1]. A subsequent NBS proposal to NSF [2] stated that 'it is also believed that a wide spectrum of practical help has become available in the last decade...' Objectives were refined in a Preliminary Prospectus [3] that gave details on proposed readership, topic outlines, and editorial administration. NSF expectations as a joint funding agency were stated in the 'Summary of Awards' issued by the (former) Division of Computer Research [4]:

This project is directed toward organizing a "Software Engineering Handbook." The initial effort consists of developing pilot technical material and subjecting it to a rigorous editorial process in the form of "Technical Notes" before it appears in a more formal bound edition. The process of refining the scope and content, of identifying knowledgeable authorities and contributors as well as finding available sources for support and publication are included in this initial effort. The objective of the "Software Engineering Handbook" is to provide a convenient source of software material that can be used in the design of quality software.

This volume. This FORTRAN volume presents material that will assist in understanding FORTRAN 66 and its replacement, FORTRAN 77. Because even a superficial comparison of the two language variants will involve contrasting their respective syntaxes, a set of FORTRAN 66 grammar rules is included: These rules, expressed in chart form, are comparable to rules that define FORTRAN 77. Next, there are two chapters of observations on what using standard FORTRAN 66 implies, and how the 1966 Standard is often interpreted and stretched to achieve practical ends. Finally, a comparison of the new FORTRAN 77 with FORTRAN 66 shows how the language has changed, and what converting older programs must entail. The four chapters address programmers concerned with FORTRAN conversions, managers engaged in programming standards, and other practitioners interested in system influences upon languages. Since the text touches upon several general programming aspects (input/output, storage allocation and lifetimes, control structures), the volume's appeal will extend beyond the immediate FORTRAN community.

Gordon Lyon, Coordinating Editor.

[1] Stewart, S.L. (Ed.) *Report on Planning Session on Software Engineering Handbook.* NBS Technical Note 832, (Nov. 1974), 18 pp.

[2] NBS Proposal to NSF, (April 1974). Later as Grant DCR75-045443.

[3] *A preliminary prospectus for a software engineering handbook.* Second Printing, (August 1975), 23 pp.

[4] *Summary of Awards: Fiscal Year 1975.* Division of Computer Research, National Science Foundation, Washington, D.C. 20550

Contents

ANS FORTRAN X3.9-1966 Charts

M. D. McIlroy

Syntax charts for FORTRAN 66 are given to aid comparisons of the older language with FORTRAN 77.

Key Words: ANS FORTRAN X3.9-1966; syntax charts; Standard FORTRAN.

1. INTRODUCTION

These charts define the syntax of ANS FORTRAN X3.9-1966[1,2,3]. Most compilers for large machines are based on this syntax, often with extensions. *[In addition, the charts served as a basis for the syntax charts of the revised standard, FORTRAN 77—Ed.]*

The charts have been designed primarily as a reference document, not as a prescription for recognition or parsing. The names of syntactic categories have been chosen to suggest terms used in the American National Standards. (One notable exception is 'element', which the standard calls 'variable or array element'.) Complete cross reference lists are included.

The charts define FORTRAN constructs in a style that is self-evidently equivalent to Backus Normal form as used to describe ALGOL, helped occasionally by footnotes. Strings of light face letters and properly imbedded hyphens denote syntactic categories. Bold face letters and other characters are literals. Certain syntactic issues have not been dealt with (with the result that the grammar becomes formally ambiguous):

—Use of blanks

—Comment and continuation lines

—Precedence of operators

—Rules about the content of DO ranges

—Relationship between declarations and uses of identifiers

—Data type matching

—Argument association

—Requirements for existence and uniqueness of labels

The undefined primitives, categories 1-5, have the following meanings:

letter: any character from the set ABCDEFGHIJKLMNOPQRSTUVWXYZ

digit: any character from the set 0123456789

character: 'Any character capable of representation in the processor' [1], which must include all letters, digits, the characters)+-*/(=.,$ and blank

line-break: the beginning or end of a FORTRAN initial line with all its continuation lines, or of an end line; comment lines are absorbed into line breaks

column-7: the beginning of a statement proper; column-7 absorbs the blank or 0 in column 6 of the initial line of that statement

Few actual FORTRAN compilers conform exactly to this standard, even as a strict superset. However, deviations are often small, e.g., forbidding 0 in places like these:

```
READ(0)
A(0*I+1) = 1.0
```

The charts owe their genesis to R. Morris's early attempt at writing down the syntax combined with the 'railroad track' idea that appeared in some Burroughs Algol charts. I wish to thank Messrs. Morris, W. S. Brown, B. W. Kernighan, A. D. Hall, and an unidentified referee for stylistic suggestions and careful help in proofreading; but responsibility for any errors substantive or clerical is wholly mine.

2. REFERENCES

[1] *FORTRAN X3.9-1966.* American National Standards Institute, New York, (1966).

[2] "Clarification of FORTRAN Standards—Initial Progress." *Comm. ACM. 12,* 5(May 1969), 289-294.

[3] "Clarification of FORTRAN Standards—Second Report." *Comm. ACM. 14,* 10(October 1971), 628-642.

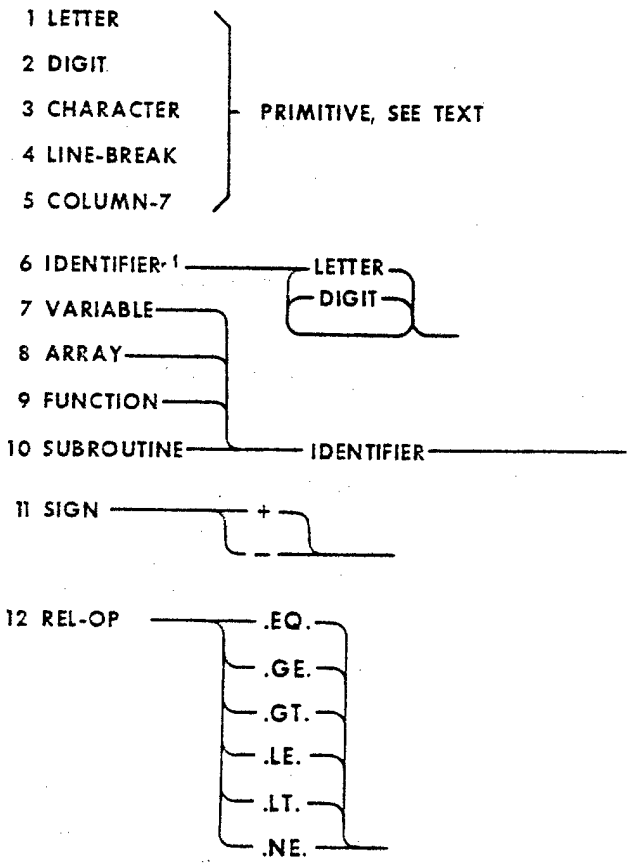
[4] *ANS FORTRAN X3.9-1978.* American National Standards Institute, New York, (1978).

3. CROSS REFERENCE INTO NUMBERED CHARTS

	46, 49, 52, 53, 55
(...)	20, 24, 25, 26, 34, 35, 36, 37, 40, 42, 43, 46, 49, 52, 54, 56

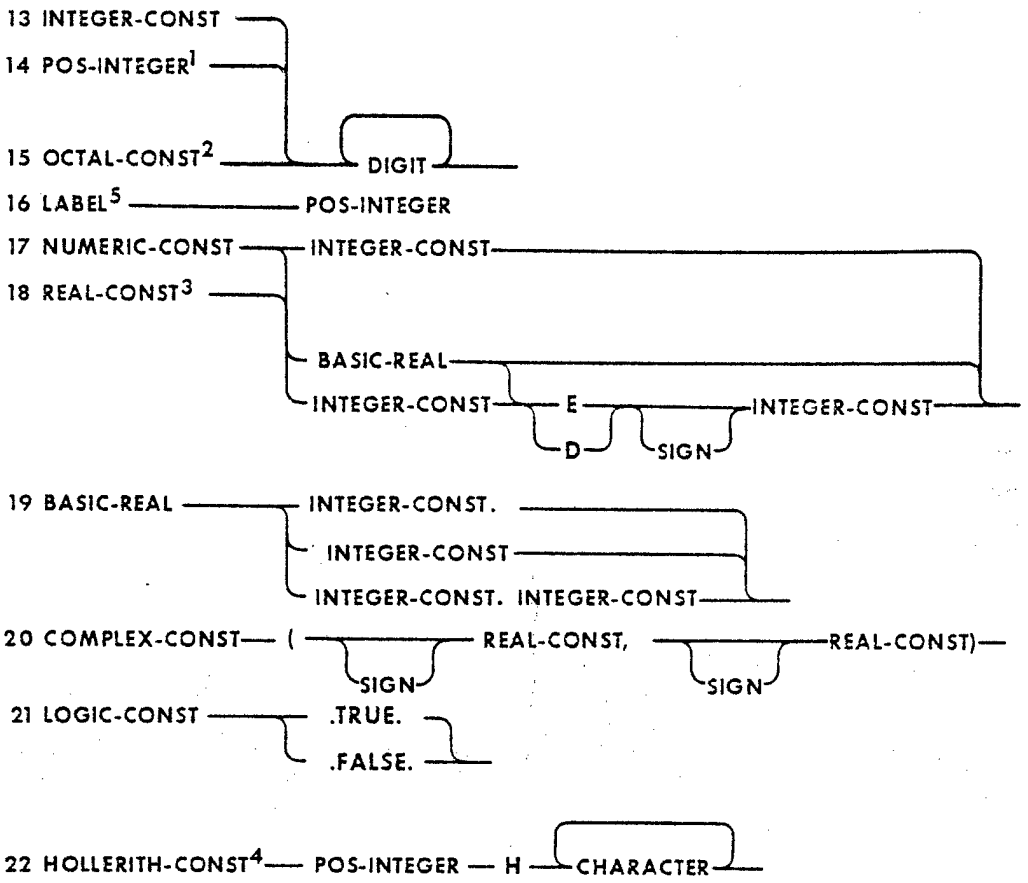
Category	*Definition*	*References*
arith-expr	39	37, 38, 40, 42
arith-primary	40	39
array	8	34, 35, 36, 45, 46, 51, 52
array-declarator	35	34
basic-real	19	17, 18
block-data	30	23
call-arg	44	37
character	3	22
column-7	5	27, 28, 29
complex-const	20	32, 40
data	31	23, 30
data-item	32	31
declarator	36	31, 34
digit	2	6, 13, 14, 15
do-specification	53	37, 52
element	46	37, 40, 42, 45, 52
end-line	29	23, 30
executable	37	23
expression	38	26, 37, 45
form	51	49
format	54	23
format-item	56	55
format-list	55	54, 56
function	9	25, 26, 34, 43, 48
function-arg	45	43, 44
function-def	26	23
function-head	25	23
function-ref	43	40, 42
hollerith-const	22	32, 44, 56
identifier	6	7, 8, 9, 10, 24, 25, 34
input-output	49	37
integer-const	13	17, 18, 19, 47, 50, 53, 56
label	16	27, 28, 37, 51
label-field	27	23, 30
labeled	28	23
letter	1	6
line-break	4	27, 28, 29
list	52	49, 52
logic-const	21	32, 42
logic-expr	41	37, 38, 42
logic-primary	42	41
numeric-const	17	32, 40
octal-const	15	37

1-12 PRIMITIVES, IDENTIFIERS, OPERATORS



NOTES: 1 IDENTIFIER CONTAINS AT MOST 6 LETTERS & DIGITS

13-22 CONSTANTS



NOTES: 1 POS-INTEGER MUST CONTAIN NON-ZERO DIGIT

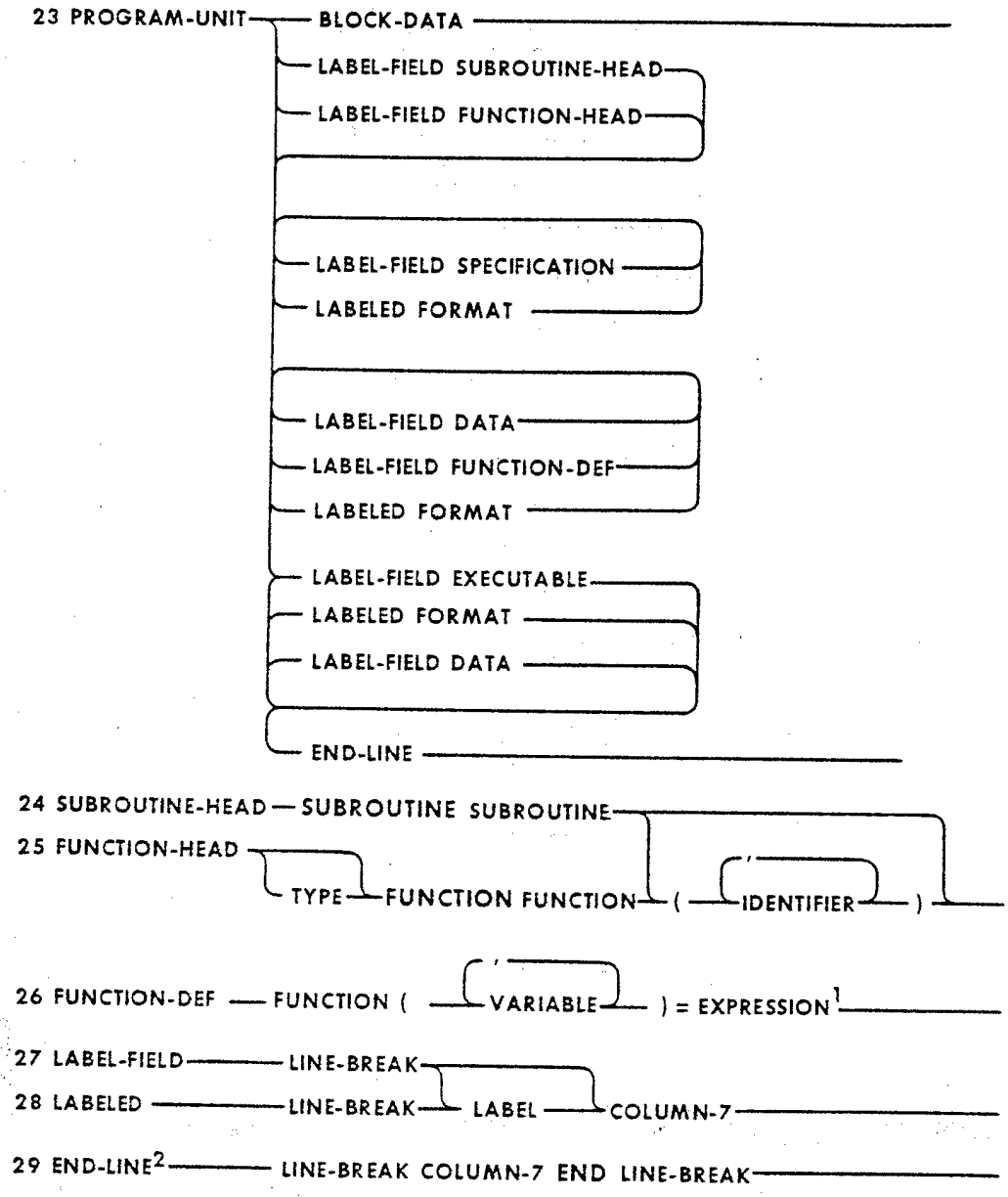
2 OCTAL-CONST CONTAINS AT MOST 5 DIGITS NONE OF WHICH MAY BE 8 OR 9

3 REAL CONST MUST NOT CONTAIN D

4 POS-INTEGER GIVES NUMBER OF CHARACTERS IN HOLLERITH-CONST

5 LABEL CONTAINS AT MOST 5 DIGITS

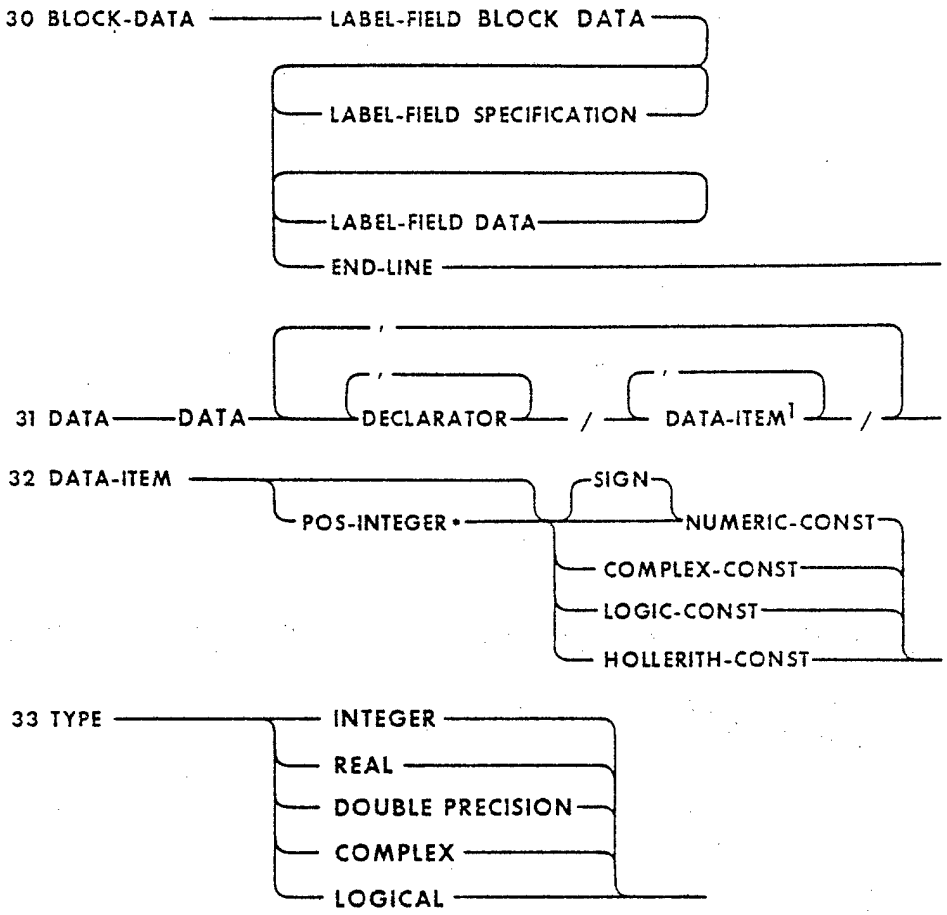
23-29 PROGRAM UNITS



NOTES: 1 EXPRESSION MUST NOT CONTAIN SUBSCRIPTS

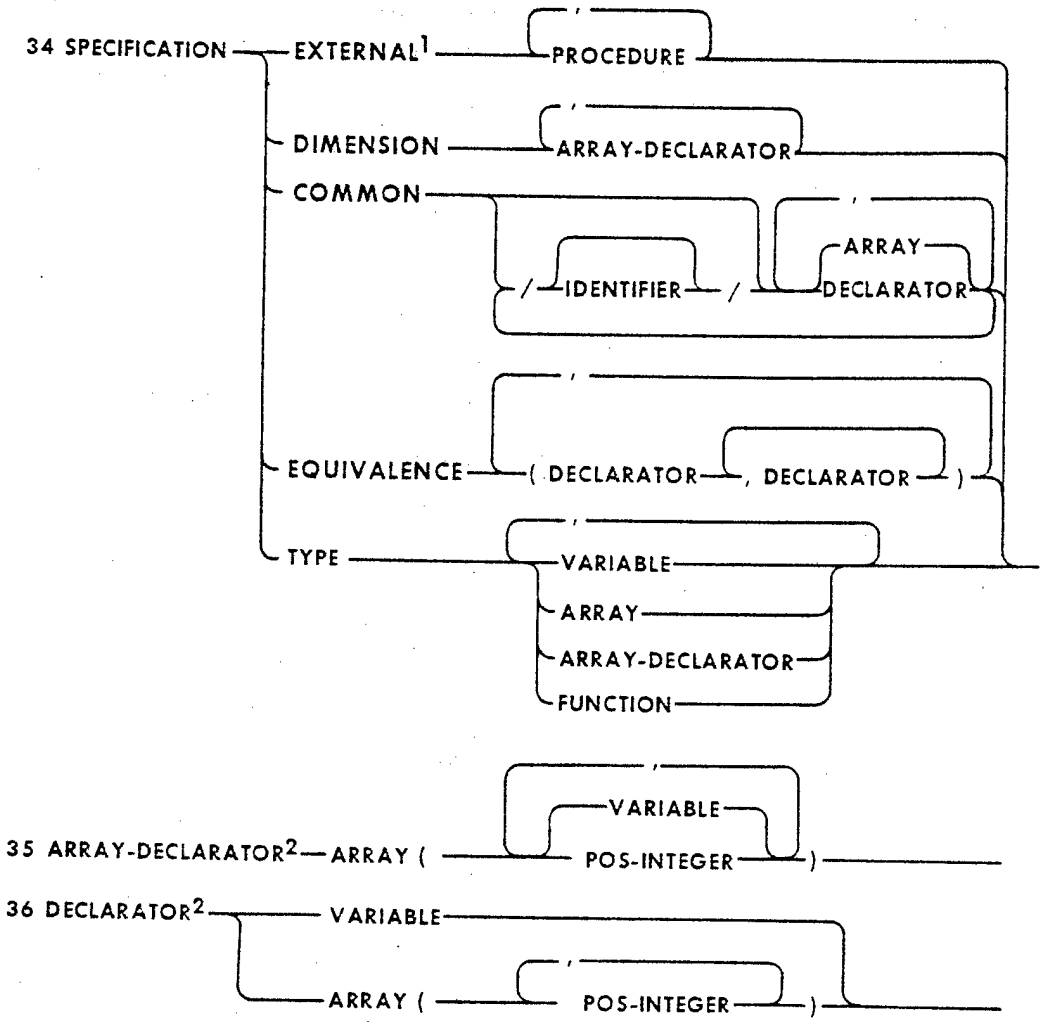
2 END-LINE MUST NOT CONTAIN CONTINUATION

30-33 BLOCK DATA SUBPROGRAMS, DATA STATEMENTS, TYPES



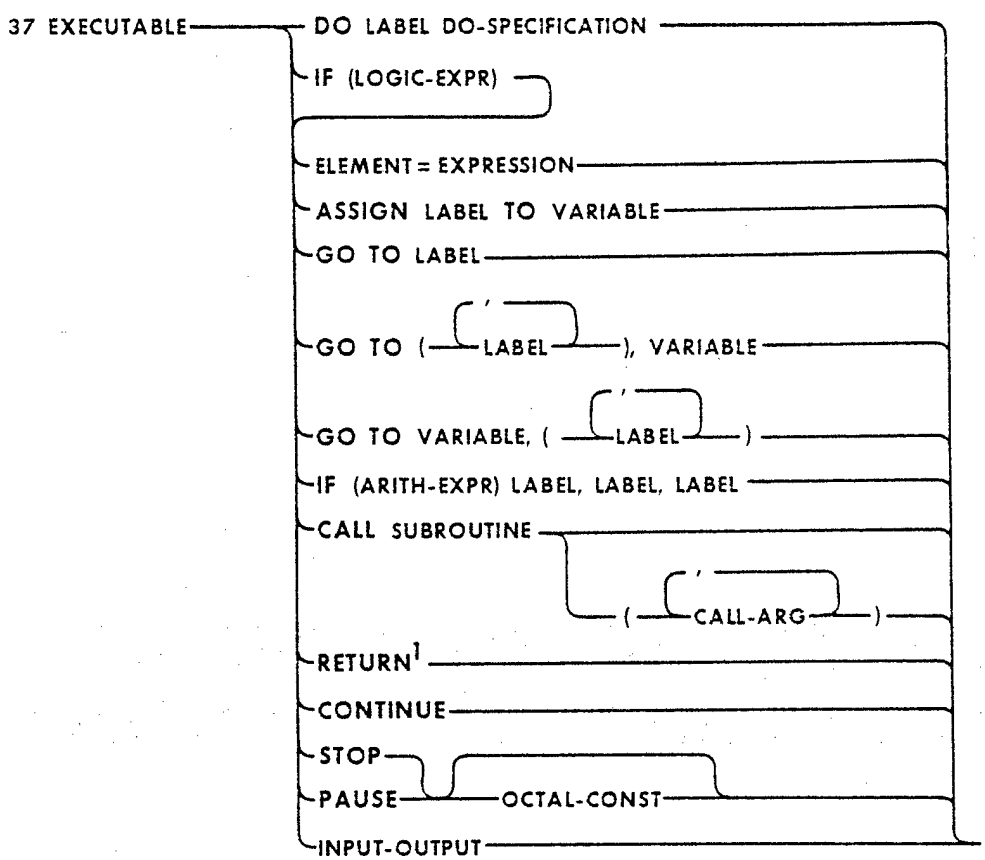
NOTES: 1 THE NUMBER OF DATA ITEMS, COUNTING REPLICATION, MUST EQUAL THE NUMBER OF DECLARATORS

34-36 SPECIFICATION STATEMENTS



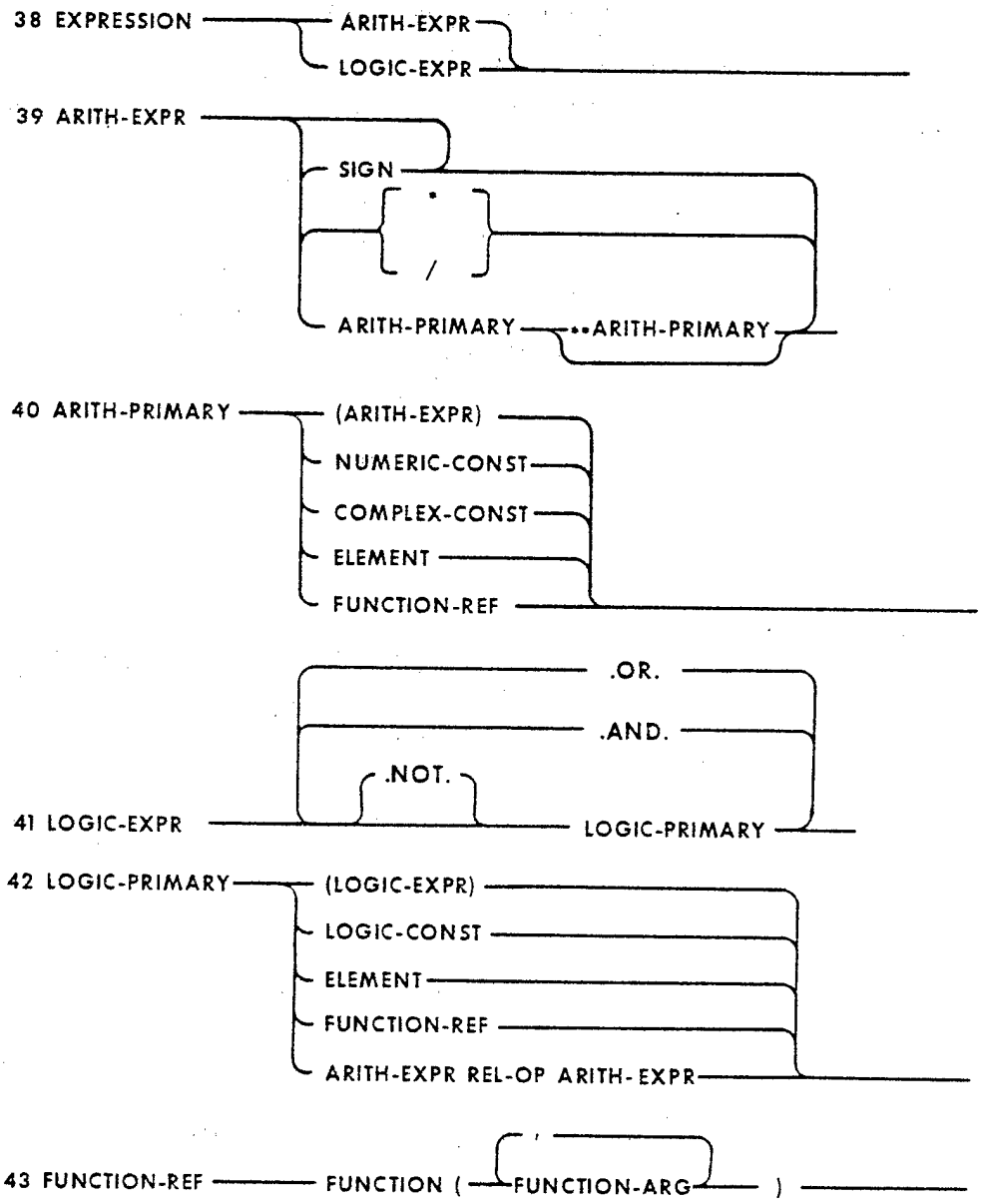
NOTES: 1 EXTERNAL IS FORBIDDEN IN BLOCK-DATA PROGRAM-UNIT
2 ARRAY-DECLARATOR OR DECLARATOR CONTAINS AT MOST 2 COMMAS

37 EXECUTABLE STATEMENTS

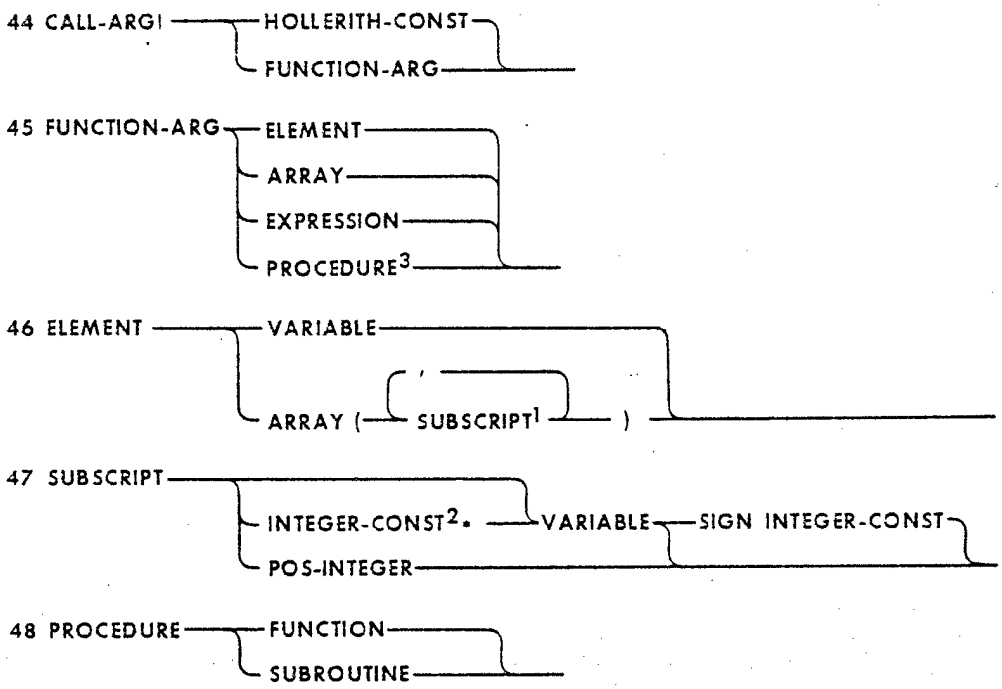


NOTES: 1 RETURN IS FORBIDDEN IN MAIN PROGRAM-UNIT

38-43 EXPRESSIONS

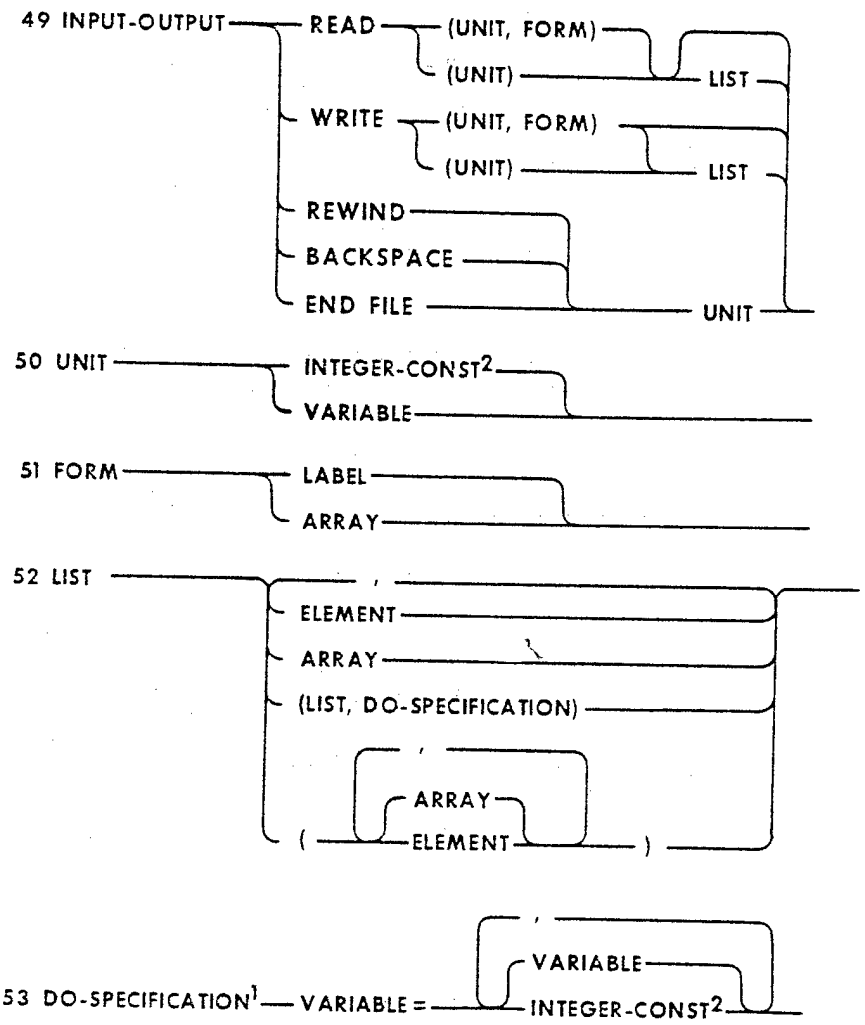


44-48 ARGUMENTS, ELEMENTS



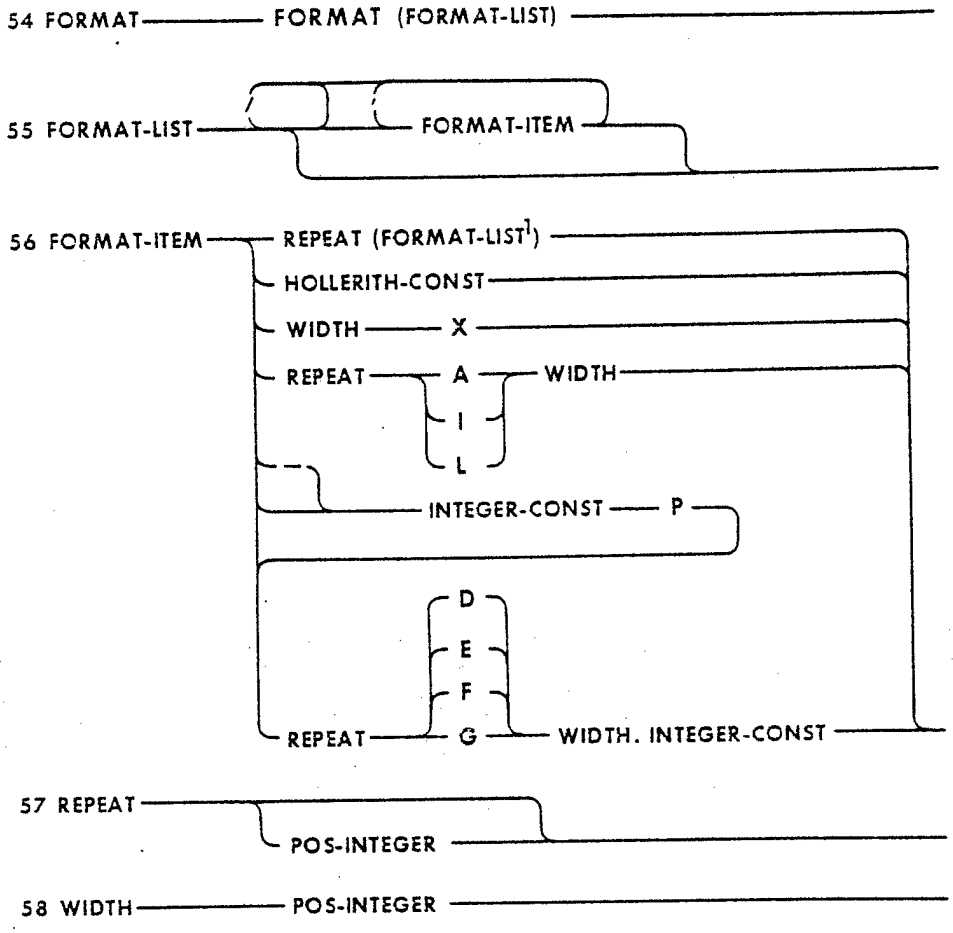
NOTES: 1 ELEMENT CONTAINS AT MOST 3 SUBSCRIPTS
2 MANY IMPLEMENTATIONS REQUIRE POS-INTEGER
3 ONLY EXTERNAL PROCEDURE MAY APPEAR AS FUNCTION-ARG

49-53 INPUT-OUTPUT STATEMENTS



NOTES: 1 DO-SPECIFICATION CONTAINS 1 OR 2 COMMAS
2 MANY IMPLEMENTATIONS REQUIRE POS-INTEGER

54-58 FORMAT STATEMENT



NOTES: 1 A FORMAT STATEMENT CONTAINS AT MOST 3 LEVELS OF PARENTHESES

Standard FORTRAN—The Rules of the Game

J. Larmouth

This chapter is addressed to people considering writing in FORTRAN (X3.9–1966). It discusses the definition of Standard (1966) FORTRAN, showing what constructs should be avoided, and what problems exist in using FORTRAN for programs which are to be heavily or widely used.

Key Words: ANSI; FORTRAN X3.9–1966; machine independence; programming languages; Standard FORTRAN.

1. INTRODUCTION

It is the purpose of this chapter to discuss the writing of 'serious' programs in a scientific environment. A 'serious' program is one which is of substantial size, carefully planned and designed to last a lifetime. Much of what is said does not apply to trivial one–off programs, except for the points that such programs grow into 'serious' programs or packages and that 'loose' programming practices are habit–forming. The chapter is *not* application oriented; there will be no discussion of algorithms for solving differential equations, performing minimization of functions and so on. The techniques and practices discussed are applicable to any algorithm, to both numerical and non–numerical work.

If a program or package (the words are used interchangeably) is to have a long life and to be of wide application in its field, it is essential for it to be easily moved from one machine to another. It used to be common to dismiss such movement with the statement, 'There is no such thing as a machine–independent program.' Nonetheless, a great many packages *do* now move from one machine to another, and the vast majority of this transfer is achieved using FORTRAN. If such a transfer can be effected with a 'reasonable' amount of work (varying from one day to twelve months, depending on the package) the package is called 'transportable.' If the transfers can really be effected with no work, the author may claim his program is 'machine–independent.' We are interested in the wider and more useful area.

For many (perhaps most) problems, FORTRAN is not the 'best' language to use. It lacks good facilities for string and character handling; it lacks recursion; it lacks dynamic space allocation; it lacks user–defined types, operators and structures. It is certainly a language of the last decade, and there are many more modern languages (and some older ones) which, from a purist's viewpoint, are 'better'. Attempts have been made to 'extend' FORTRAN in various ways, either by tidying it up with the removal of some restrictions or by grafting on to it a block–structure or adding new statements. It is, of course, in a computer manufacturer's interest to produce and to encourage the use of extensions which are peculiar to his machine. All serious programmers should completely eschew such extensions. With only a few exceptions they make little real difference to the ease (or possibility) of writing or of using a FORTRAN package.

FORTRAN was produced in the late 1950's for use on IBM computers. The first use of the term FORTRAN II was in 1958, the first FORTRAN available on an IBM 704 around 1956. It became widely available on a number of machines during the 1960's (FORTRAN IV

appeared in 1962) partly because it was backed by a large manufacturer. Its success was also partly due to a number of pronounced advantages over its main competitors—the Autocodes, largely developed in England, and ALGOL 60, a European attempt to design a 'good' language. In the early days of FORTRAN, machine efficiency was much more highly rated than it tends to be today, and programmers had to be won over from the use of machine code. The language was such that it could be efficiently compiled using the compiling techniques then available, and could be viewed by the user simply as an aid to the production of machine code. Other languages tended to present the user with 'an ALGOL machine' (for example); that is, the user tended to be unaware of how his program was compiled and run. FORTRAN also gained much from the early availability of optimizing compilers, and the ability to incorporate machine code subprograms in a tidy manner. This last feature is a direct consequence of the language design which permits the completely independent compilation of subprograms with minimum requirements for the linker or loader which combines the subprograms into a complete job, and for run time linkages. Thus libraries of FORTRAN subroutines with an easily stated specification for linkage quickly became available. In some other language systems library routines had either to be built into the system (often in machine code) or had to be physically incorporated in user source programs.

In 1962 a working party of the then American Standards Association was set up to try to produce a specification for FORTRAN. In 1964 a draft specification was produced, and in March 1966 this was finally approved. Up to this time there were a number of dialects of FORTRAN, the most important being called IBM FORTRAN II and IBM FORTRAN IV. In general FORTRAN II was a subset of FORTRAN IV (although there were exceptions) and two standards were produced by the American Standards Association (ASA). One was called FORTRAN and was closely related to IBM's FORTRAN IV. The other was called BASIC FORTRAN and was a strict subset, intended for use on smaller machines. BASIC FORTRAN was closely related to FORTRAN II in the features it provided. These standards—ASA FORTRAN and ASA BASIC FORTRAN—gradually became widely accepted, and most manufacturers will now claim that their compilers conform to the Standard. Occasional references appear in the literature to 'BASIC FORTRAN IV' or 'ASA FORTRAN II'—these usually indicate that the author is paying lip-service to the Standard, but has never read it.

In August 1966 the American Standards Association changed its name to 'The United States of America Standards Institute' and references to USASI FORTRAN began to appear. The Institute published the old ASA specification (unchanged) in a new cover. Finally, in October 1969 the name again changed to 'The American National Standards Institute' and references to ANSI FORTRAN appeared. Throughout the remainder of this paper we shall refer merely to Standard FORTRAN—meaning that Standard first laid down by ASA and continued unchanged by USASI and ANSI. (See references [2] and [3] for more details.)

The past few years have seen a lot of activity on FORTRAN Standards. A new Standard has been produced and widely circulated [4]. (Chapter Four of this volume has details on it.) Like the original, the 1978 Standard (called 'FORTRAN 77') is an attempt to codify existing developments in FORTRAN, rather than to design a new language. Nonetheless, some element of the latter is inevitably involved. There will be a period in which few FORTRAN compilers truly comply with the new FORTRAN 77. The rest of this Chapter deals entirely with *1966 FORTRAN,* which is a *de facto* Standard, very widely accepted. (All references to 'the Standard' mean this old Standard.) The new Standard is not immediately in that category, and programmers should in the first instance treat it with care. Much of this Chapter will need to be re-written when (or even if) we reach the state that all FORTRAN compilers for large

computers conform to the new Standard, but until that time, the qualifications and restrictions mentioned in what follows should be adhered to.

It is important to note at this point that the Standard in no way prevents a compiler writer from extending the language. A compiler conforms to the Standard provided it accepts at *least* the statements defined in the Standard; furthermore, there is no requirement concerning the handling of 'incorrect' programs, that is, programs containing statements not defined by the Standard. While this means that in practice all compilers accept a slightly different language, it has helped to gain acceptance for the Standard and to help it achieve its stated object of 'promoting a high degree of interchangeability of .... programs.' The reader can be reasonsably sure that if his FORTRAN program conforms to the Standard in all respects, and provided he takes certain other precautions to be outlined later, then his program will run on a wide variety of large general purpose computers.

For a precise definition of the Standard see reference [1].

2. STANDARD FORTRAN

In this section attention is drawn to some of the features of the Standard specification. This specification has been described as 'a careful documentation of the bugs in the earliest FORTRAN compilers.' A 'minimal work' approach, particularly with an eye on producing efficient code without elaborate analysis of the source program, is likely to generate much the same restrictions in present day compilers. Later comments will make this clearer. With the 1966 Standard, there are few constraints on the compiler writer, and consequently many on the user who wants his programs to be portable. (The 1978 Standard of FORTRAN 77 has moved somewhat towards freeing the user, and constraining the compiler writer, but the effect remains to be seen.)

Most machine instructions require a reference to some storage location which is to take part in an operation. Thus A=B will require a reference to the location B followed by a reference to the location A.

These references are usually of a quite restricted form. Most machines possess storage locations (addressable memory), and one or more registers. A reference is usually made up of some storage location, together with an offset which is obtained from some specified register, sometimes called an index or a modifier. The storage locations can be taken in what follows as numbered from 0 upwards, so that if the FORTRAN integer I is normally held in location number 50, and an array A starts at location number 100, a reference to A(I) would be compiled as

Set a register from location 50

Reference, modified (or indexed) by the register,

location 100.

Thus each such reference takes two instructions. Now if we consider a series of FORTRAN statements referring to A(I), B(I), C(I), etc., it is clearly wasteful to load the register from I each time—the compiler simply loads it once, and then *assumes* I is held in the register. Thus we now have two resting places for the value of I—location 50, and the register. Hopefully these will never get out of step— whenever anything happens to change location 50 (e.g. by setting the tenth element of an array K which starts at location 40 and which was equivalenced by EQUIVALENCE (I,K(10)) the compiler will notice, and reset the register. In practice these two resting places for I *can* get out of step unless the compiler is very well written, and/or is prepared to insert many more instructions than are strictly necessary in most cases. This is one of several possible examples where alternative approaches to code generation will produce different results for certain programs.

Some compilers will accept that they must produce bad code in dangerous conditions, e.g., when any equivalencing has been done. Others will so completely analyze the program that they get it right. However, both these approaches are costly, and the cheap and nasty solution is to tell the programmer not to do anything which can fool a simple-minded compiler. The Standard attempts to point out those programming practices which will (or at least, which in the past) fool(ed) simple-minded compilers. With this background, we can now proceed with the Standard.

There is no substitute for a careful reading of the Standard itself. We attempt here to clarify the meaning in some difficult areas, and to highlight some points which might be missed. However, for many people, learning to write in Standard FORTRAN is an unlearning process, and what has to be unlearned depends very much on the individual background. There follows a selection of features you must *not* use when writing a Standard program.

(1) Mixed mode expressions.

(2) Arrays with more than three subscripts.

(3) General integer expressions in subscripts. The correct rule is Constant * Variable + (or -) a Constant.

(4) FUNCTION A(I) followed by INTEGER A.

(5) IMPLICIT and NAMELIST.

(6) RETURN instead of STOP in the main program.

(7) It must not be assumed that

A+B+C

will be done as (A+B)+C rather than as A+(B+C). (Note that these are different; the associative law does not hold for floating point arithmetic.)

(8) A**-B is not allowed.

(9) Implied DO loops in a DATA statement are not allowed.

(10) With A real, C complex,
C = C+A is permitted but not
C = A

(11) With D double precision, A, B real,

D = D+A(I)*B(I)

may or may not form the
double length scalar product.

(12) There is no mechanism for
supplying a label as an
error exit on a CALL statement.

(13) No REAL*8, etc. statements
are permitted.

2.1 Available Statements

Assignment statements

Complex expressions can only be assigned to complex variables and logical expressions to logical variables. Otherwise any type can be 'coerced' (to use an ALGOL 68 term) into any other type.

ASSIGN

It is illegal to copy and restore an integer variable between the ASSIGN and the ASSIGNED GOTO statements. In particular links cannot be stacked.

GOTO

There are three forms—the unconditional, the assigned and the computed.

Arithmetic and logical IF's

IF(.....) —,—,—,

IF(.....) .....

CALL statement

RETURN statement

CONTINUE statement

DO statement

This is discussed later. Note, however, that the controlled variable cannot cross zero or go backwards, and it is illegal to attempt to obey the loop zero times.

PAUSE

This statement is difficult to implement in any sensible and efficient way on modern multiprogramming computers. Many compilers which otherwise conform to the Standard will unashamedly treat PAUSE as STOP. Do not use it.

STOP

READ and WRITE, BACKSPACE, REWIND, ENDFILE

Note that the Standard does not define what unit values are available, nor does it assert that BACKSPACE, REWIND and ENDFILE should be applicable to all units.

Both formatted and unformatted READ and WRITE are allowed, and the Standard allows them to be mixed. However, the programmer is well advised to do no such mixing.

Note that there is no 'Direct Access I/0,' nor is there any provision for trapping errors or end–of–file.

DIMENSION

COMMON

EQUIVALENCE

These three, DIMENSION, COMMON, and EQUIVALENCE, are discussed later.

EXTERNAL

The specification should be read carefully to discover when an EXTERNAL statement should be used. Most compilers will allow rather more freedom than the Standard, provided the program is not actually ambiguous.

REAL, INTEGER, DOUBLE PRECISION, COMPLEX, LOGICAL

A later section, *3.0 SPECIFIC PROBLEM AREAS,* discusses the precision of arithmetic in more detail, but for the present, note that the Standard merely requires DOUBLE PRECISION to give 'a greater precision than REAL.' However, it also compels twice as much computer storage to be used for holding these variables as is used for REALS. In practice, although it is not forced by the Standard, DOUBLE PRECISION variables usually have a precision which is at least twice that of REALS.

DATA

Note the lack of an implied DO loop in the DATA statement. This is one area where the restrictions of the Standard make programming much more tedious than it is with the extension. The possible undefinition of variables which are initially defined in the DATA statement is discussed later.

FORMAT

The following items can appear in a FORMAT statement:

F,E,G,D,I,L,A,H,X,P

and no others. Note that one or more /'s and comma are used as delimiters, but that /,/ is not strictly allowed, although many systems will permit it.

Statement Functions

The dummy arguments can be safely used for other purposes within the subprogram. In one definition it is possible to refer to other statement functions only if they have been defined earlier: this completely prevents recursive use of these functions. Finally, note that the right hand side of an arithmetic statement function is not a general expression, since it may not contain references to elements of any array.

This is the first example of keeping things simple for the compiler. An arithmetic statement function is usually made much more efficient than a normal function call, so that the compiler does not normally want to 'tidy up' before calling one. 'Tidy up' means, among other things, ensuring that if the storage location for I is not up to date, the value being currently held in a register is dumped in the storage location.

The most usual case when the register is more up to date than the storage location is when the variable is acting as the control variable of a DO. It is likely on some compilers to be held in a register throughout the DO, and only put in the storage location if a jump out of the DO occurs. Thus

```
      DIMENSION B(4)
      F(A) = B(I)+A
      DO 1 I = 1,4
      C = F(D)
    1 CONTINUE
```

is liable to fail with that sort of simple-minded compiler. The Standard therefore says, simply, that array references may not appear.

Available intrinsic functions

```
ABS, IABS, DABS
AINT, IDINT, INT
AMOD, MOD
AMAX0, AMAX1, MAX0, MAX1, DMAX1
AMIN0, AMIN1, MIN0, MIN1, DMIN1
FLOAT
IFIX — prefer INT to IFIX in
       general; INT is defined to give
       the same result as assignment
       of the real value to an
       integer.   IFIX is merely
       required to produce some
       integer value, and is there-
       fore somewhat undefined for
       non-integer values.
SIGN, ISIGN, DSIGN
DIM, IDIM
SNGL
REAL
AIMAG
DBLE
CMPLX
CONJG
```

Note the lack of DINT—taking the integer part of a double precision variable is only possible if the value is small enough to be held exactly in an INTEGER or a REAL. Theoretically, the effect of

```
D = DINT(D)
```

can be obtained by using the basic external function DMOD (see later):

```
D = D - DMOD (D, 1.D0)
```

However, as the DMOD has to be formed as the (small) difference of two large numbers, loss of precision can result unless the DMOD is very carefully coded. In practice, if DINT is needed, a machine-code routine may be needed.

(Note that intrinsic functions, in contrast to basic external functions, cannot be redefined by a user subprogram, unless they are given a different type. It is sometimes dangerous to rely on the letter of the law, and a safe policy is never to use the names of basic intrinsic or external functions for other purposes unless it is absolutely essential.)

Basic external functions

```
EXP, DEXP, CEXP
ALOG, DLOG, CLOG
ALOG10, DLOG10
SIN, DSIN, CSIN
COS, DCOS, CCOS
TANH
SQRT, DSQRT, CSQRT
ATAN, DATAN
ATAN2, DATAN2
DMOD
CABS
```

Note that COSH and SINH can be obtained by using EXP, and ACOS and ASIN by using ATAN.

Observe also that it is not defined where the cut occurs in the complex plane for CSQRT and CLOG, so that the result provided by these functions can vary by multiplicative factors of e**(i*π/2) and additive factors of i*π respectively. The safest policy is to normalize the results of these functions yourself to the ranges you require. The same problem occurs with ATAN.

t FUNCTION, SUBROUTINE (t=REAL, INTEGER, etc.)

A subprogram must have a RETURN somewhere in it, although it may in fact finish on a STOP. Hollerith strings may be supplied as arguments for a CALL statement, but are not permitted in a function call.

BLOCK DATA

Any package which is to be used in the form of a library of routines, some of which may be used and some not for any particular application, cannot make effective use of the BLOCK DATA subprogram, for there is no means of referencing it, and hence of obtaining conditional loading. In practice, for all but complete programs, it is better to rely on the user to make an initializing call than to use BLOCK DATA.

3.0 SPECIFIC PROBLEM AREAS

3.1 Sizes

This section deals with the number of repetitions of things, with the magnitudes of numbers and with the precision of variables. The Standard lays down that a compiler must accept the following.

(1) Up to nineteen continuation lines.

(2) Three-dimensional arrays.

(3) Two levels of parentheses in a FORMAT statement.

The Standard does *not* specify the length of subprograms, the number of arguments which are allowed, the lengths of common blocks and so on.

The only rule to follow is to be moderate—an argument string on one line is reasonable; a subprogram a few pages long is reasonable. Much more than this, and trouble can be expected on some machines.

More important is the maximum size of a real number accepted by any machine (for most machines the maximum size of DOUBLE PRECISION variables is about the same as REAL variables). Another important number is the maximum value of integer which can be held exactly in the machine— either in INTEGER, REAL or DOUBLE PRECISION variables. It will always be found that larger integers can be held more exactly in DOUBLEs than in REALs and it is sometimes the case that REALs will hold larger exact integer values than INTEGERs. Note that the maximum size of number that can be held at all is usually *much* larger than the maximum integer value that can be held exactly.

Reasonable values to assume are about 10**50 for maximum values, about 10**4 for integers in INTEGERs, 10**6 for integers in REALs or 10**12 for DOUBLEs, but these figures will vary very much from machine to machine. If you are even approaching numbers of this general size you should carefully document your program with the maximum values you expect to need. If you ever come close to stretching your own machine to its limits, and you wish your program to be transportable, you should consider reprogramming with some built-in scale factors.

The precision of REAL and DOUBLE PRECISION variables is a closely related topic. Floating point arithmetic is not exact, even when relatively small numbers of places are involved.

E.g. A = 1.23*100.0

will not set A to exactly 123 on many machines. This is because 1.23, although it is a terminating decimal, does not terminate when expressed in binary; thus an error is incurred in holding the number in the first place. On the other hand,

A = 1.25*100.0

would be likely, on many machines, to set A to exactly 125.

The above point should not of course be relied on, and was introduced for illustrative purposes only. The basic assumption should be that *all* real constants will be entered with a small error equal to the precision of the machine, and that all operations will be performed on this erroneous data, and will then introduce a further error.

There is a large field of study which is concerned with producing algorithms for widely needed tasks which minimize this error build up. If one is using a published algorithm for a standard process (e.g. inverting a matrix) it is likely to give reasonable results on most machines. If the problem is ignored it is likely that while rough answers can be obtained on one machine, on other machines with less precision, the answer is completely swamped by error. Whenever there is any danger of problems of this nature the precision, base and method of rounding performed by the machine on which the program was tested should be carefully documented.

In general, if a local library routine is used (e.g. for matrix inversion), it is better not to embed it solidly in the program, but to make it easy to use an equivalent routine available on another machine.

A final point in this connection is the question of IF statements on REALs or DOUBLEs. Because an expected zero will almost always be perturbed to one side or the other, tests of the form

```
IF (A) 1,2,3
```

should be avoided. In general the program should be written so that

```
IF (A) 1,1,3
```

and

```
IF (A) 1,3,3
```

will produce the same final result. Where this is not possible, the test is best made in the form

```
IF (ABS(A).LT.1E-6) GOTO 2
IF (A) 1,2,3
```

where the value 1E-6 is chosen to suit the particular problem.

Since this is not intended to be a text on numerical analysis, the discussion will not be taken further than to warn of the possible dangers. It is precisely because of this problem that people say 'there is no such thing as a machine-independent program.'

3.2 Hollerith Strings

The most important point to note about Hollerith strings is that they are a rather unhappy addition which is not very well defined. There is no particular type of variable for holding such strings; the operations which are possible on variables holding them are ill-defined; the length of string which can be held in a variable is not defined. In particular, Hollerith literals may only appear in the DATA, FORMAT, and CALL statements; they cannot appear in a function call, and there is no mention of what the dummy arguments should be like in a subroutine whose actual argument is a literal string. (This area is addressed and much improved in the new draft Standard. However, the 'improvement' does not reflect a widespread practice. There is at present little common ground between compilers beyond that of the existing Standard.)

It is best to use only A1 format for input or output, and 1H in the DATA statement. Literal strings in the CALL statement should be avoided except when calling machine-code routines. It is defined that an A1 output from a variable set in the DATA statement by 1H will work and produce the desired effect. However, copying or comparison of variables containing strings is not explicitly stated to be possible. Copying of REALs by, e.g. A=B will, on many implementations, result in the bit-pattern in B being 'standardized' or 'normalized'— the essential point is that, interpreted as characters, A will not be set to B. Thus Hollerith strings are generally more safely held in INTEGERs. (When we look for comparison, we find that some machines will give integer overflow when comparing two INTEGERs—despite this, INTEGERs are probably on balance the safest bet for holding characters.)

It would be unacceptable for many programs to refrain from the use of these items, and such advice would be unrealistic. However, the programmer should in general treat any comparison or copying of Hollerith strings as an extension to the language, to be carefully isolated and documented.

3.3 DO Loops

The specification goes to some trouble to define the ways in which DOs and their CONTINUEs can interleave, but the rule is the 'obvious' one. Similarly, there are restrictions on what the terminal statement of a DO may be. These restrictions can be summarized by saying that a statement which may cause a transfer of control is not permitted.

If you note that

```
      DO  3 ......
      ...
      ...
3     GOTO 7
```

is treated by compilers as a shorthand for

```
      DO '3' ......
      ...
      ...
   3  GOTO 7
  '3' CONTINUE
```

it is clear that the CONTINUE can never be reached, and the program is likely to be in error. Remember that the specification does not say such programs must be diagnosed, merely that they lie outside the Standard and may be diagnosed.

A slightly less obvious and more easily violated restriction is the rule that if several DOs share the same terminal statement, there may not be a transfer of control to the statement from anywhere other than inside the innermost DO. Once again, the reason for this restriction will be made clear by an example:

```
      DO      3  ......
      ...
      ...
      ...
      GOTO    3
      ...
      ...
      ...
      DO      3  ......
      ...
      ...
      ...
      GOTO    3
      ...
      ...
      ...
    3 CONTINUE
```

The structure of the DO is equivalent to

```
      DO    '31'  ......
      ...
      ....
      ...
      GOTO    3
      ...
      ...
      ...
      DO    '32'  ......
      ...
      ...
      ...
      GOTO    3
      ...
      ...
      ...
    3 CONTINUE
 '32' CONTINUE
 '31' CONTINUE
```

Thus the innermost GOTO probably means GOTO '32' and will fall through to that statement. However, the first GOTO 3 probably means GOTO '31', but is in fact jumping incorrectly into the middle of DO '32'.

It would require a 'clever' (and probably expensive) compiler to treat the original as

```
      DO  '31'  ......
      ...
      ...
      ...
      GOTO '38'
      ...
      ...
      ...
      DO'32'  ......
      ...
      ...
      ...
      GOTO'39'
      ...
      ...
      ...
 '39' CONTINUE
 '32' CONTINUE
 '38' CONTINUE
 '31' CONTINUE
```

and few existing compilers do so. (Equally, this error is rarely diagnosed.)

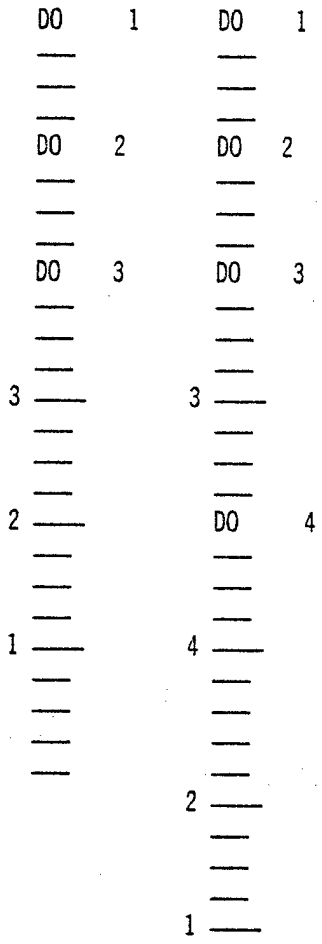
This leads us on to the general question of jumping into DO loops. There are often several hidden locations (or registers) involved in a DO loop, e.g. DO 3 I = 1,9,2 may use basic machine instructions to count around the loop five times. On some machines this might be most efficiently done by

```
Register = -4
.....
.....
.....
.....
.....
.....
If register is non-zero, increment
     it by one and jump back.
```

Another register may hold 'I', and yet another the increment. On other machines a totally different set of instructions might be compiled, but in all cases there are likely to be registers or locations which are not known to the user. This immediately implies that jumping into DO loops from outside cannot be permitted in a straight-forward system, and makes it likely that jumping out and *then* back again will be fraught with danger. The rules in this connection involve the concepts of 'a completely nested nest' and the 'extended range' of a DO.

A completely nested nest is a collection of DOs, not contained in any other DO, and not containing further DOs, such that the first 'end of a DO' occurs after the last DO.

Thus, the first column is a completely nested nest, while the second is not.



The Standard specifies that a compiler, to conform to the Standard, must accept a jump out of a DO and back in again to the same DO, at least when the jump occurs from within the innermost DO of a completely nested nest, right out of that nest. Some compilers will work correctly with jumps out and in under more general circumstances, but for a program to conform to the Standard a jump out and in is only permitted as described above. The set of statements obeyed during the time spent outside the DO before jumping back in is called the extended range of the DO. It is permitted to have other DO statements in the extended range of a DO, but none of these may have an extended range. In general then, extended ranges are messy things, and need to be treated with care. (Note that these points in no way prevent a jump out of a loop if there is no intention of jumping back in again.)

3.4 Arrays, Equivalence, etc.

There are two ways of looking at the effects of equivalence, common declarations, and argument passing. In this section we will view these activities in a way closely corresponding to

their actual implementation, which will bring out the reason behind many of the more straightforward prohibitions in this area. Later, when we look at association, definition and undefinition, we will view the effect of these statements in a more theoretical way.

For the present, then, we view a particular labeled COMMON block as a single piece of storage. In different routines the locations of this storage may be given different names, and may be grouped in different ways to form arrays. Essentially this is merely giving different names to the same locations. Equivalencing can now be viewed as simply giving another, alternative, name for certain locations or groups of locations. With this viewpoint, most of the rules given under COMMON and EQUIVALENCE in the specification make good sense — only constant subscripts allowed — no equivalencing of two COMMON items and so on. (Notice that the Standard very carefully specifies the way in which the elements of a two- or three-dimensional array are mapped into consecutive storage units, and that items adjacently declared to be in COMMON will be placed end to end in memory.)

It *is* permitted to exceed the bounds of an array subscript, provided only the bounds for the whole array are not violated, e.g. DIMENSION A (3,4) will permit reference to A (11,1) which gets the same storage location as A (2,4). Reference to A (13,1) is not permitted, even if it is well defined what that location should be, e.g. by

```
DIMENSION A(4,3), B(13)

EQUIVALENCE (A(1,1), B(1))
```

However, there are a number of compilers that will check that individual subscripts do not exceed their bounds, and the above freedom should not be exploited.

The passing of arguments is best viewed as happening in two steps. In the calling routine, the address of the argument (its location in memory) is calculated. In the called routine, either a reference to the dummy argument will contrive to use this address, or if the dummy argument is a simple variable, the value of the argument may be picked up and copied into a location provided for the dummy argument, and at the end of the routine copied back again. Note that we once again have the value of a variable held in two places at once: its original position (which may be in COMMON) and the dummy argument hidden location; we can expect problems and restrictions related to this.

For the present, we note a few simple restrictions which are fairly obvious from the foregoing overview:

(1) Equivalence operations must not be performed on dummy arguments.

(2) A dummy argument may only be written to if the actual argument is a simple variable or subscripted array name.

(3) If an actual argument happens to be in a COMMON block, neither the argument nor the COMMON block location may be written to.

Finally, we note one freedom which is widely used, and which all serious compilers will allow. This is the ability to pass across the start address of some storage locations by giving the array element which corresponds to the first, and using those locations in the called routine as an array. The dummy argument array may not extend beyond the end of the array being passed, but the dimensionality may be totally different. This is most often used for performing operations on the columns of a 2-D array. Suppose we wish to use a routine MAX which finds the largest element of a one-dimensional array, and to apply it in turn to each column of the 2-D array B. Then something like the following is both permitted and widely used:

```
      DIMENSION B(10,20)
      DO 1 I = 1,20
      CALL MAX (B(1,I),10)
      ...
      ...
1     CONTINUE
      END

      SUBROUTINE MAX (A,J)
      END
```

3.5 Classes and Scopes of Names

The Standard takes up one and a half pages describing the classes (I to VIII) of a name, and defining which such classes have meaning globally to a program, and which are of local meaning within a subprogram. The rules do not differ from those required by almost every compiler written for FORTRAN, and the reader should have no trouble in understanding this section.

In summary, the identifiers used for actual subprogram names, for basic external functions, and for the labels of labeled COMMON form a group of names which, if used in two different subprograms, mean the same thing—they are global. We call these 'global' names. All other names are local to a subprogram, and if used in some other subprogram, there is no connection between the two. We call these 'local' names.

There are four points to note. Firstly, the use of EXTERNAL puts the following name into the global category, although almost all compilers in fact do this only if the name is not a dummy argument.

Secondly, the same name can be used in a global manner for a common block, and also locally, within the same routine, thus COMMON/A/A is permitted. With this exception, if a name is referred to for global use in a program unit, it cannot be used for local purposes in the same unit.

Thirdly, the name of a function is available as a local variable in its own body, as well as giving the name globally to the function.

Finally, the names of the intrinsic functions are effectively local—they can be used for the intrinsic functions in one subprogram, and be given global meaning as a user function or labeled common name in another routine.

4. STORAGE ALLOCATION IN FORTRAN

There are essentially two main techniques for allocating storage to a FORTRAN program. Compilers exist which use both, and *both* techniques produce a system conforming to the Standard. The two techniques are here called static and dynamic allocation. The former is the most common by far, and a number of programs advertised as conforming to the Standard rely on static allocation and circulate widely.

We have from time to time talked about a certain variable being, for example, at location 50 in memory. The question is, why location 50, how was the number obtained? For the purposes of storage allocation there are essentially four sorts of entity:

(1) Local variables, and hidden locations needed for DO loops or evaluating complicated arithmetic expressions. These are the 'LOCALS' for the program unit.

(2) The blocks of labeled COMMON referred to by the routine.

(3) Unlabeled COMMON.

(4) The actual code and constants of the program unit, called 'CODE'.

Most complete programs will have one unlabeled COMMON, several labeled COMMON blocks and several program units each with CODE and LOCALS.

Suppose that we have a program consisting of a main program, three subroutines S1, S2 and S3, and three COMMON blocks B1, B2 and B3. B1 is referred to by the main program, B2 by S1 and S2, B3 by S3 only. Assume further that the routines are presented in the order Main, S1, S2, S3. Then in static allocation of core, the place occupied by these items is fixed once and for all before the program starts to run, and will look something like the following (note, however, that small variations on this theme are possible).

```
Main        CODE      Bottom of storage
Main        LOCALS
Block       B1
S1          CODE
S1          LOCALS
Block       B2
S2          CODE
S2          LOCALS
S3          CODE
S3          LOCALS
Block       B3
Unlabeled   COMMON
                      Top of storage
```

We see that CODE and LOCALS for a routine are loaded as the routine appears. Labeled COMMON blocks are loaded after the first routine referring to them, and unlabeled COMMON goes at the end. (It will now be clear why a labeled COMMON block must have the same length in all routines, why unlabeled COMMON need not and why local arrays cannot have varying dimensions.)

Before considering the implications of this approach, we will discuss the dynamic allocation method. In this case, we have rather more scope for variation. The simplest scheme is described first. Suppose the flow of control at run-time passes as illustrated in Figure 1. That is, Main calls S1, S1 calls S2; S1 then calls S3; then return to Main which calls S2.

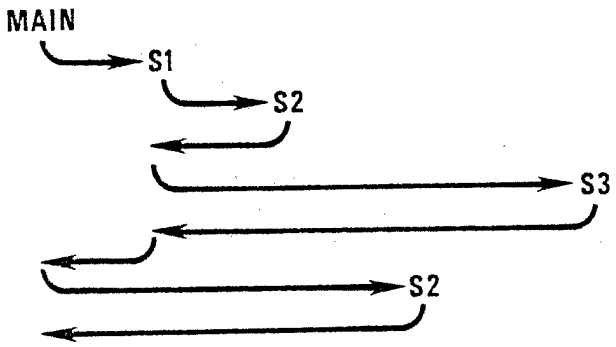


Figure 1. Flow of control

We assume two totally separate areas from which memory can be allocated. We put in the first area all CODE and unlabeled COMMON. This is fixed once and for all at the start of the program.

Main	CODE
S1	CODE
S2	CODE
S3	CODE
Unlabeled	COMMON

The second area is used as a stack. Consider the progress of the stack as the job runs.

(a)	Enter Main. Allocate space for the Main LOCALS and for block B1 on the stack.	Main LOCALS Block B1
(b)	Call S1. Allocate space for S1 LOCALS and for block B2 on the stack.	Main LOCALS Block B1 S1 LOCALS Block B2
(c)	S1 calls S2. Allocate for S2.	Main LOCALS Block B1 S1 LOCALS Block B2 S2 LOCALS
(d)	S2 returns. Delete S2	Main LOCALS

	LOCALS (i.e. clear memory, or perhaps leave the values there, but consider it available for allocation).	Block B1 S1 LOCALS Block B2
(e)	S1 calls S3. Allocate for S3.	Main LOCALS Block B1 S1 LOCALS Block B2 S3 LOCALS Block B3
(f)	S3 returns to S1, then S1 returns to Main. Delete S3 and S1 and their corresponding blocks.	Main LOCALS Block B1
(g)	Main calls S2. Note that Block B2 is in the same position in the summary opposite, but it is not in the same position in core, for in general S2 LOCALS will not be the same length as S1 LOCALS.	Main LOCALS Block B1 S2 LOCALS Block B2
(h)	S2 returns and Main returns. All storage is freed.	

Now that the basic principle is understood, we mention very briefly some variants. First of all, one area of memory can be made to suffice if the length of unlabeled COMMON is calculated (by the loader). Alternatively, the stack can work down memory towards the CODE and unlabeled COMMON can work up towards the stack. A more far-reaching variation is to use dynamic allocation for the CODE as well—that is, to keep the CODE on backing storage, and to load the CODE for a routine onto the stack at precisely those points in the above description where the LOCALS were allocated. The reader will notice that in the fully dynamic approach, less memory is used than in the static approach.

When a routine (S1 say) calls another (S2 say), S1 will usually set the return address in a fixed register. S2, on entry, will normally dump this register in a hidden location in the LOCALS. Thus S2 can call S3 by setting the register in the same way. On exit, S2 can pick up its return address from the hidden location.

We are now in a position to understand the rules set out in the Standard. The most important point to note is that the Standard in no way discriminates between the two methods of implementation. If a program is written to conform to the Standard, it will run completely satisfactorily with either sort of implementation.

Let us consider the static approach first, and the restrictions it imposes. We have already noted that all storage is allocated before the program runs, so arrays cannot have varying sizes. The second point to notice is that if a routine S1 calls S2, and S2 then calls S1, the same locals and hidden locations will be used for the second call of S1 even though they are still in use for the first call. Thus, if this second 'activation' of S1 returns to S2, and S2 then returns to the first activation of S1, the local variables will have been disturbed by the call for S2. More particularly, the hidden location containing the return link for exit from the first activation of S1 will have been overwritten by the return link (to S2) of the second activation. Thus if the first activation now returns, it will incorrectly go to S2 which, on a RETURN will do as it did before and go to S1, which will go to S2, etc. The attempted calling of a routine by itself, albeit indirectly, is called recursion, and is prohibited by the Standard.

(Note that it would have been possible for the above processes to occur provided the second activation of S1 never returned. A few packages are in circulation which rely on this. S1 is a steering routine, which will read a word from an input stream and decide which of several processing routines to call. It never obeys a RETURN, ending on STOP when the input is exhausted. The processing routines may call other routines, but at the end of processing, the return to S1 to read more data is not by RETURNs all the way back to S1, but is by a simple call of S1. This might be called semi-recursion. It will work on many compilers, but is equally outside the Standard.)

Now let us look at dynamic allocation. Most of the restrictions imposed because of static allocation are no longer necessary. If a local array is given a dimension which is passed as an argument to the subroutine, this will cause no trouble. The length of the array can still be calculated in time for storage allocation to take place (because the LOCALS are only allocated on entry to the routine). Secondly, we note that if a second activation of a routine occurs, it will be natural to simply provide a second set of LOCALS on the stack, and all is sweetness and joy. Recursion goes through. However, suppose we now look at the semi-recursion, which worked before. We now find that, as routines rarely if ever obey a RETURN, the size of the stack grows continuously, and the job very quickly collapses for lack of storage.

The third point to note is the effect of S1 calling S2, S2 setting a value in a local variable and returning, and then S1 calling S2 again. What is the value of the variable? In static allocation the value will be what it was set to on the previous entry, and this combined with the DATA statement to initialize the value before the first entry provides a very tempting facility. Variables with this property are usually termed *own* variables.

Suppose we now look at the dynamic approach for this job. The locations used for the LOCALS of S2 will normally be completely scratched on exit from it, and the next entry will have a fresh set of (possibly zeroed) variables. Similarly, any COMMON blocks which are referred to by S2, but not by S1 or any routine above it, will get allocated after the LOCALS for S2, and will also be 'refreshed' by a RETURN and a re-entry.

Variables given initial values in the DATA statement pose a problem. These can be placed in the CODE part of the routine, and will then act like *own* variables; but note that they will then be common to all activations of the routine. Alternatively, various expedients can be used to copy initial values into LOCALS on the stack on each activation (or into COMMON BLOCKS to cope with BLOCK DATA). The Standard cuts the Gordian knot and simply says

that if a variable is initially defined by a DATA statement and never written to, it remains defined for all activations of the routine. If, however, it gets written to, or is not initially defined in the DATA statement, the compiler writer is free to do what he likes on exit from the routine, and the programmer has violated the Standard if he assumes that the changed value will be retained. To summarize:

(i) Most compilers use static allocation.

(ii) The Standard allows implementors to use either; it imposes enough conditions on the programmer to ensure that a program will run and produce the same results whichever system is being used.

5.0 DEFINITION AND UNDEFINITION, ASSOCIATION, SECOND-LEVEL DEFINITION

This is the last aspect of the Standard to be looked at. Many people seem to find it the most difficult, but we have already met one of its rules in the previous section when we discussed the DATA statement:

'...An entity...may be initially defined' (by the DATA statement) '..and remains defined until it becomes undefined' (i.e. an exit from the routine occurs). 'The redefinition of an initially defined entity can result in a subsequent undefinition of that entity.'

In ordinary English, if a local variable is initialized in a DATA statement and is never written to, then it will have that same value on every entry to the routine. If, however, somewhere in the routine it is written to, then on the next exit it will lose its value, and on the next entry its initial value is not defined (it may be the DATA statement value, it may be the value written to it, depending on the implementation, but the Standard allows either).

This example should help the reader in his interpretation of the phraseology used by the Standard.

When discussing COMMON, EQUIVALENCE and argument passing, it was noted that these activities could be seen as simply providing different names for the same storage location. In this section we must view these activities in a rather different, more theoretical, light.

Whenever a common block is mentioned in a subprogram, it has a number of pigeonholes (or storage locations) allocated to it. If two subprograms declare the *same* block (even with the identical structure and local names) we must visualize two sets of locations, but these are linked (Figure 2).

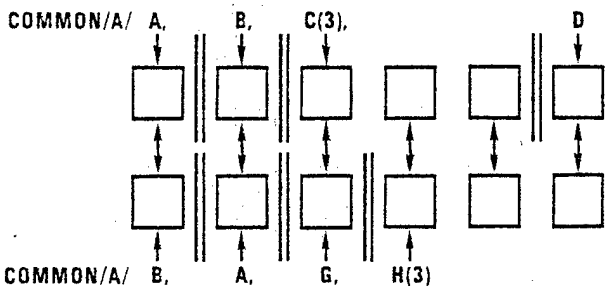
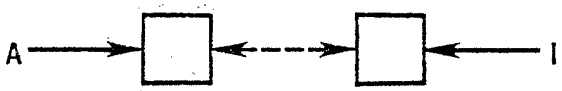


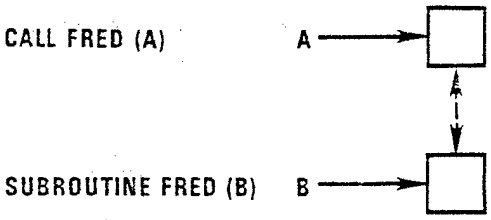
Figure 2. Association caused by COMMON declarations

In Figure 2 a two headed arrow represents the association between array element C(2) in the first routine and array element H(1) in the second. How this association is implemented is irrelevant to the present discussion.

Similarly, if we have an EQUIVALENCE statement EQUIVALENCE (A,I) on two local variables, we do not consider A and I as sharing storage. We consider two separate locations A and I, with an association between them.

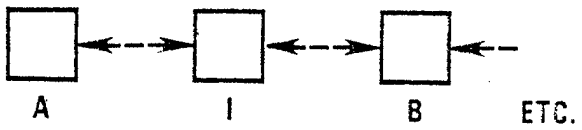


So far the association has been a static one, set up at compile time. However, we can consider the actions of a CALL statement as setting up an association between a variable in one routine, A, say, and a variable in another routine, B:



The action of RETURN is to break this association, and *then* to undefine B.

The association may connect several variables, in the same or different routines, in COMMON or locals, of the same type (REAL, INTEGER, etc.) or different.



Using this notation, the Standard then gives various rules about assigning values to entities. Two of these rules are given here.

If an entity becomes defined (gets a value), then any associated entities of differing type become undefined, and those of the same type become defined. Note in particular that if an INTEGER and a LOGICAL entity (variable) are associated, writing a one or a zero to the INTEGER is *not* guaranteed to produce a TRUE or FALSE in the LOGICAL. The LOGICAL value is undefined, thus leaving the representation of truth values entirely in the compiler writer's hands.

The second rule says that if an entity appears in an input statement, it is assigned a value as the statement proceeds. However, associated entities only take on this value at the end of the input statement. Thus:

```
EQUIVALENCE (I,J)
READ I, (A(K),K = 1,J)
```

does not conform to the Standard.

```
CALL A(K,K)
SUBROUTINE A (I,J)
READ I, (A(K),K = 1,J)
```

is similarly in error.

Next, let us look at the simple question of when we can and cannot define (i.e. assign to, roughly speaking) an entity.

(1) An entity cannot be redefined if it is an argument of the subprogram and is associated with another argument, or with an entity in COMMON, e.g.

```
COMMON A
CALL F(A)
END
SUBROUTINE F(B)
COMMON A
B =                    is illegal
A =                    is illegal
END
```

```
CALL G(A,A)
END
```

```
SUBROUTINE G(B,C)
B =                    is illegal
END
```

(This is another case of restrictions to avoid problems arising from having a value in two places. In some implementations writing or reading the argument will actually access the common location. In others it will merely access a local variable which is initialized on entry and copied to the common on exit. In yet others it will depend on the type [INTEGER or REAL] of the argument.)

(2) All entities are initially undefined unless they are defined in a DATA statement.

(3) The following entities cannot be redefined.

(a) The control variable, terminal parameter or increments of a DO while in the range or extended range of that DO.

(b) A variable dimension.

(c) A parameter of a function call which is used elsewhere in the statement.
e.g. A = F(A) is illegal if F assigns A.
B = A+F(A) is obviously illegal
and so is CALL PRINT(A,F(A)) —
a more common case.

(d) If the actual parameter for a dummy argument is a constant or expression, the argument cannot be redefined.

Entities become undefined in a number of circumstances, some of which have already been mentioned. There is no substitute for a careful reading of the Standard, but here are two:

(1) The control variable on completion of a DO is undefined.

(2) The expression A.LT.B.OR.C.LT.FN(D) where FN(D) assigns to D leaves D undefined (because it may not be necessary to enter the function at all in order to evaluate the expression).

We should also mention the concept of 'an entity being defined on the second level' or 'second-level definition.' So far, we have been talking about an entity being defined on the first level.

Second-level definition is concerned with the use of an integer variable in a subscript. We noted earlier that when an integer was used for a subscript it would often have to be picked up into a register before being used, and that it might be useful to hold it semi-permanently in the register. The Standard lays down some rules saying when an integer variable becomes defined or undefined on the second level and imposes the constraint that it may only be used in a subscript if it is defined on the second level. This allows optimizing compilers to produce the 'wrong' answers in awkward cases. Second-level definition applies only to integer variables, and the rule is that if it is to be used either in a subscript or in a computed GOTO, then it must be defined for use on the second level.

In order to specify when an integer is defined for use on the second level, the Standard introduces the concept of a *basic block*. A basic block is terminated by a DO statement, a CALL statement, any statement which transfers control, an assignment statement which assigns to an INTEGER variable or a READ statement which does so. A block is also terminated before any statement which an IF or a GOTO may jump to. Thus we see that, roughly speaking, an integer variable cannot change its value within a basic block. The major exception to note is a function with an integer argument which assigns to its argument.

The Standard then states that an integer variable is defined for use on the second level (i.e. in a subscript or GOTO) if it is defined at the first level on entry to the basic block, and if it does not become redefined or undefined on the first level during the block. The second level definition corresponds roughly to the value, picked up into a register at the head of a basic block, and used throughout that block. Thus if a new value appears in the storage location, the register will be 'wrong' and the compiler has produced the 'wrong' code. The Standard says that one must not cause this to happen.

Thus, suppose a function F will return a value, and also set its single integer argument to 1, 2 or 3 to denote a completion code, one might be tempted to write

```
A = F(I)
GOTO (1,2,3),I
```

but this violates the Standard.

There is one other important rule here. If an integer variable becomes redefined, all associated variables in the same program unit are undefined for use on the second level throughout the rest of the program unit, unless they are explicitly redefined. Thus:

```
EQUIVALENCE (I,J(10))
READ (—,—)K
J(K) = 4
...
...
...
B(I) = 3
```

is perfectly valid, provided the value read in is not 10!

Finally, we should note that there are some exceptions to the above rules for input/output statements, to allow statements like

```
    WRITE (—,—)(A(I),I = 1,5)
or
    READ (—,—) I,A(I)
```

to be used, and guarantee their correct interpretation.

This completes the survey of the Standard. This final section on second level defintion has perhaps justified the comment 'a documentation of the bugs in early compilers,' and indeed, most compiler writers today would feel that they should do what the user intended in many of the 'undefined' or 'forbidden' examples quoted above. However, it is very easy to avoid exploiting such features once you are aware of their existence, and it is better to err on the cautious side where large packages are concerned.

6. USING FACILITIES OUTSIDE THE STANDARD

In this section we discuss the precautions which should be taken when it is *necessary* to exceed the Standard. Any extensions used should be thought about carefully, and only put in if there is no other way of doing the job. It is, however, equally silly to take the opposite view and to assume that extensions, local routines or assembly code can *never* be used. Three specific cases in justification of this viewpoint are given.

Firstly, there are the direct-access I/O statements available on some FORTRAN systems. These permit a random access to any record of a data file, and are an essential feature for some types of application. Many machines have such a facility, but the precise syntax for it differs from machine to machine. Of course, use of REWIND and a read through the data file to the required position would, in theory, do the job: in practice it would never be considered. If the application can be reworked to avoid the need for random access, so much the better; if not, its transportability has been limited (e.g. to machines with disks) but not totally precluded if care is taken.

Secondly, we consider programs which require a pseudo-random number generator. This is the type-example of a routine which, although it may be written in a high-level language, usually contains constants which are very machine-dependent. If the 'randomness' requirements are not serious, and considerable loss of efficiency can be accepted, it is possible to formulate a routine which will work on all machines with a 'reasonable' word length, but the two hypotheses do not always hold. Furthermore, almost all general purpose installations have available one or more random number generators which have been tailored to the machine. A similar example, mentioned earlier, is a routine to invert a matrix.

Thirdly, consider the problem of taking two integers I1 and I2, forming the number I1*IB+I2 and dividing it by I3, a third integer. This is the mate of the problem of forming the number I1*I2 in the form I3*IB+I4 where IB is some value which can just be held in an integer, and I1, I2, etc. are any values less than IB. (IB will, of course, be set to suit the machine.) These two problems occur in the writing of a package for performing arithmetic on arbitrarily large integers, IB being the 'base' of the number system used. If it is kept sufficiently low that IB**2 can be held, there is no problem, but the space needed to hold large numbers is doubled; otherwise, very considerable contortions are needed to produce a Standard FORTRAN routine which will do the job. On the other hand an assembly-code routine (or a routine using some local facilities—e.g. REAL*16 on an IBM machine) can usually be produced to do the job very easily. (Note that the Standard does *not* specify that a DOUBLE PRECISION variable can hold exactly an integer equal to the square of that which can be held in an INTEGER—often it cannot.) In this case the FORTRAN routine should be provided, together with a recommendation that it should be recoded.

Let us now consider some extensions which *might* be used in a program. There are two main groups. In the first group we find routines which ease the job of writing a program, but make no real addition to the facilities available; however, they *are* available on a reasonable number of compilers. The following two items fall into this category:

(1) The use of mixed mode expressions, e.g.

```
A = A + I
```

This does, of course, obscure the cost of such a statement while

```
A = A + FLOAT (I)
```

makes it more explicitly expensive; however, this is not the point here.

(2) The use of the quote symbol to delimit Hollerith strings thus avoiding the need to count characters: 'THE MIND IS INFINITE' is easier to produce than 20HTHE MIND IS INFINITE. This is only really of any importance in FORMAT statements, as long strings should be avoided elsewhere.

There are a number of extensions which make programs easier to write that do not fall into this category (i.e. are not universally available). We mention only three:

(1) There are various extensions available to the DO loop—in particular crossing zero, and going in a negative direction.

(2) Use of names longer than six characters.

(3) The IMPLICIT statement which, although very useful, is not essential.

This last group should of course be completely avoided, but the reader may well be tempted to use items in the first group mentioned. In general, this action is not advised, but if there is a firm intention never to export the program, some back-sliding might be permitted. But note that

(a) A number of 'small machines' are now, by dint of much effort, being provided with compilers for full Standard FORTRAN. Although some extensions may be provided by the giants in the computer field, there are many good solid systems, less well known, which do not have the extensions.

(b) These items will usually be detected at compile time on a system without the facility, and any competent programmer can make the change. (However, your package may well get a bad name, and you may soon find a rival version, with the extensions removed, in circulation.)

(c) It really is not much harder to write a program without such props!

The next group to consider consists of facilities which add to the language and make possible the writing of certain programs that could otherwise not be written. .

Examples are:

(1) Direct access I/O (discussed above).

(2) Traps for the end of file on input.

(3) Copying and comparison of Hollerith strings.

(4) Free format input/output (otherwise known as list-directed I/O).

Items in this category are very often available, but the precise form can change. Furthermore, there will be a number of systems which will not have the facility.

The rule here is that where such extensions are essential, they should, as far as possible, be collected into one single routine, and a careful write-up provided of what that routine does. Where this is not possible, a very noticeable pattern, e.g.

```
Cxxxxxxxxx
Cxxxxx
Cxxx
Cx
```

should be put around the offending item to make for easy changing later. The advertisement of the package should make it clear that these extensions to the Standard are needed for running the program.

Next, we will consider the use of local library routines. (Two points mentioned under extensions—trapping and free format I/O— are often provided as library routines.) Many local routines are concerned with interfacing to a particular type of operating system— opening files, setting the clock, reading time and date, and so on. In general these should be avoided; if you really want to print out date and time, make it a self-contained routine which does nothing as supplied, but which the user can alter to print date and time on some specified unit. Sometimes it is useful to leave in your own local calls and associated instructions (with a C in column 1), to indicate how to go about writing the routine.

Other local routines perform operations which are widely available— matrix inversion, Runge-Kutta differential equation routine, Gaussian integration, sorting a two-dimensional array by some specified row or column, random number generator, and so on. In these cases the routine may be available only in machine code form, or may well contain constants which are strongly related to the local machine. The routine may even be copyright protected and cannot be moved to another machine! Whenever possible in these cases, you should provide a Standard FORTRAN routine which does the job, but make sure you document it carefully if it is suspect and contains machine-dependent features. The most important point is to provide a detailed specification of what the routine is supposed to do, of every point in the program that calls it, and of the calling sequence assumed. Whoever takes the package on some other machine can then replace it with one of his own if one exists, or modify it if necessary. If it is possible to write a Standard routine which is no more machine dependent than the rest of the program, then it would be best to replace the local library routine by that.

Finally, we come to the thorny subject of assembly language sections. If these are essential, then keep their specification simple. If at all possible, provide an equivalent FORTRAN routine, even if it will be much slower or give much less precision. It may well be a guide to the production of a better routine, or it may suffice altogether if your package is to be only lightly used. In some cases a well annotated machine-code routine can be a help in writing a similar routine for other machines, but do not overdo it! While machine code can have its place for obtaining facilities and sometimes for coding inner loops, it is rare for it to be justified in a program which is to circulate.

This chapter has attempted to lead the reader through some of the potential pitfalls in writing FORTRAN programs, and in so doing to make him more aware of the problems involved in writing a program designed to last (i.e. to be transported). Lest depression has set in, I must add that a great many good programs do circulate. Avoiding the pitfalls is perfectly possible, provided the will is there, and the Standard specification is carefully studied.

7. REFERENCES

[1] *FORTRAN X3.9-1966.* American National Standards Institute, New York, (1966).

[2] J.E.Sammet *Programming Languages: History and Fundamentals.* Prentice-Hall, Englewood Cliffs, N.J. (1969).

[3] "History and Summary of FORTRAN Standardization Development for the ASA." *Comm. ACM. 7,* 10(October, 1964), 590-625.

[4] *ANS FORTRAN X3.9-1978.* American National Standards Institute, New York, (1978).

[5] J. Larmouth. "Serious FORTRAN" and "Serious FORTRAN—Part Two." *SOFTWARE: Practice & Experience 3,* 2-3(1973), 87-108, 197-225.

Standard FORTRAN—Writing the Program

J. Larmouth

This chapter is a follow-up to the earlier Standard FORTRAN—The Rules of the Game. *Although the present chapter can be read on its own, it is best read after the earlier chapter has been mastered and the reader has a clear understanding of what statements constitute Standard FORTRAN X3.9-1966 and which are extensions to it. This chapter assumes throughout a thorough knowledge of the 1966 Standard. A variety of topics are covered, and many of them are topics which are relevant to any programming language. In some cases the solutions proposed would also be applicable to other languages, but the chapter has been written with emphasis firmly on the FORTRAN programmer. The aim is to help the reader write a* good *FORTRAN program that nonetheless conforms to the ANS FORTRAN X3.9-1966 Standard. The sections can be read in any order and cover the main areas determining the quality of a program.*

Key Words: ANSI; FORTRAN X3.9-1966; FORTRAN extensions; standard FORTRAN.

1. DYNAMIC SPACE ALLOCATION

We have noted in the earlier chapter that, in order to allow static allocation of space by the system, FORTRAN does not permit

```
SUBROUTINE MATRIX (A,N)
DIMENSION A(N,N),B(N,N)
```

yet this is very often needed to obtain workspace in a routine.

Three solutions to this problem are in fairly wide use.

(1) A local array of sufficient size for the largest N is used:

```
DIMENSION A(N,N),B(10,10)
```

and the user is told N must not exceed (in this case) 10.

(2) The user is told to provide an array himself; this means he can, if he wishes, carve it out of some larger array. So we have

```
SUBROUTINE MATRIX (A,B,N)
```

It also means that, even if he sets it as a fixed dimensioned local array in his main program, the fixed size can be better geared to his likely use than when it was built into the routine. The disadvantage is that the user interface has been made a little more messy.

(3) The routine uses the first locations of unlabelled COMMON for B. This means, firstly, that the user must beware of using COMMON himself, and secondly, that a non-standard facility (usually available) is being used. The technique is to declare a one-dimensional COMMON array of vast size (too large to fit in core) but to use only part of it; or alternatively to declare it to be of size 1, and exceed the bounds; thus we get

```
SUBROUTINE (A,N)
DIMENSION A(N,N)
COMMON//B(1)
or
COMMON //B(100000)
(The former is more common.)
```

Note that the B's two-dimensional nature must be handled by the program. There is a refinement possible here if the user can be persuaded to discipline himself. This is possible where he is using a large package rather than an isolated routine. The first location of unlabelled COMMON is used to hold the current position in COMMON for the start of work-space. This is used on a stack basis - if a routine wants space it simply increments the pointer. If a routine called by it wants space, it again uses the pointer position. Thus the user in his main program may have statically allocated

```
COMMON//A(10,20),B(50),C(600)
```

He changes this to

```
COMMON//IPTR,A(10,20),B(50),C(600)
IPTR = 851 (Setting IPTR to the first
     location of the free space above
     the static arrays)
```

Imagine a call with real parameters for array A1(N,N):

```
CALL MATRIX (A1,N)
```

The subroutine contains:

```
SUBROUTINE MATRIX (A,N)
COMMON//IPTR, B(1)          (or B(100000))
DIMENSION A(N,N)
ISTART = IPTR               (Preserve pointer
IPTR = IPTR + N*N           (Allocate space
CALL MATRI1(A,N,B(ISTART))  (Call 'real' routine
IPTR = ISTART               (Restore pointer
RETURN
END
SUBROUTINE MATRI1(A,N,B)
DIMENSION A(N,N),B(N,N)
 ....
 ....
 ....
 ....
END
```

Note that this is using the completely standard facility of declaring B with a totally different dimensionality in the called routine from the one it was given in the calling routine.

The only non-standard feature is the COMMON//...B(1). The whole process can be made standard by replacing the 1 by the largest integer such that the COMMON will then fit in the available store. However, on many implementations either tactic will work.

2. RECURSION

This section discusses problems which arise when trying to code (in FORTRAN) an algorithm in which a function calls itself, possibly indirectly. This is commonly called *recursion.* We have already seen in the earlier chapter that the Standard forbids recursion, so some work will be needed to handle the problem.

The simplest and most often quoted example of recursion is the definition of *factorial n* as

```
F(n) = 1 if n = 0
     = n*F(n-1) otherwise
[n integer, non-negative]
```

In a language which permitted recursion we would write:

```
REC FUNCTION F(N)
F = 1.0
IF (N.EQ.0)RETURN
F = F(N-1)*FLOAT(N)
RETURN
END
```

In FORTRAN we do not have this facility and so the whole definition would usually be turned around, and we would use the iterative form

```
      FUNCTION F(N)
      F = 1.0
      IF (N.EQ.0) RETURN
      DO 1 I = 1, N
1     F = F*FLOAT(I)
      RETURN
      END
```

If a recursive call takes place, we have first of all the overheads of a function call (often large) and secondly the extra memory needed for all the LOCALs of F for this second call. Thus many people would assert that, even were REC FUNCTION F available, the second form should be preferred for any function where it is possible.

Roughly speaking, if the values of the arguments needed can be easily spotted, and if the function values for these arguments can be used as they are produced, then an iterative

approach can be found. However, if each definition of F results in a variable number of calls to F depending on some data which is either global or an argument, then full recursion is usually needed. An example is the problem of finding all faces of a polytope in n dimensions given the corners (a polytope is a generalization of a polygon-polyhedron).

Consider a polyhedron, defined by its corners, and suppose we know the set of corners forming one face, and on that face the set of corners forming one edge. The given corners enable us to set up a linear programming matrix which 'describes' the polyhedron. Using this, we can start with the known edge, and roll round each of its (two) corners on to the neighboring edge in the known face. Proceeding in this way we get all edges of the face. We can now take our known face, and with each edge in turn roll over to the neighboring face (this is the same routine we used for finding the edges, but one dimension higher up). We now have some new faces, (each with a known edge), and we apply the entire procedure to these, resulting in more (and eventually all) faces. The number of times the procedure is applied, both for finding neighboring edges in a face, and for finding neighboring faces, depends on the geometry of the body.

In n dimensions, given an n - 1 dimensional subspace (a 'face'), and an n - 2 dimensional subspace (an 'edge'), we provide a routine which finds all 'faces' (n - 1) by first finding all 'extremal points' (n - 3) of the n - 2 spaced 'edge'. This is a recursive call. We can then roll to neighboring n - 2's in our n - 1, and then to neighboring n - 1's through each n - 2. Thus we finally have all 'faces' (n - 1) of the original n-dimensional polytope. Although totally different algorithms can be invented for this job, it is not possible simply to reshape it as an iterative method as we did for factorial n.

The above problem is too difficult and complicated to be useful as an example of how to obtain recursion if you need it, but it serves to show that not all recursive problems can be trivally changed into an iterative form. A fuller discussion of recursion is found in [1], and [2] may be of interest to anyone attempting to find optimum solutions to games and puzzles, perhaps the most common application of recursive programming.

As a compromise between triviality and complication, consider the function F defined by

```
F(M,N) = F(N,M) for M<N
F(M,N) = M if N = 0,
       = M*F(M/N,N - 1) + N*F(M - N,N) for M>=N, N>0
```

where M and N are positive integers, and M/N means integer division. (Mathematical readers might care to re-form that into an iterative algorithm!)

In our recursive language we have

```
      REC FUNCTION F(M,N)
      IF (N.EQ.0) GOTO 1
      IF (M.LT.N) GOTO 2
      TEMP=F(M/N,N-1)*FLOAT(M)      A
      F=TEMP+F(M-N,N)*FLOAT(N)      B
      GOTO 3
1     F=M
```

```
      GOTO 3
2     F=F(N,M)                    C
3     RETURN
      END
```

Now in Standard FORTRAN (and most dialects) we have to do this recursion the hard way by programming our own stack and links. First let us see what happens on a RETURN. Control may:

(1) Exit totally.
(2) Return to complete statement A.
(3) Return to complete statement B.
(4) Return to complete statement C.

Next we must consider the values held in variables. When F calls itself, we talk about a second 'activation' of F. (This term is used because the original F is still 'active' in the sense that the values of some of the variables will still be needed, and must be restored when we return to continue after this call of itself.) What variables do we need to keep belonging to an old activation before beginning a new one? It is clear we need M, N and TEMP, but F is not needed. We here point out a device that has already been incorporated. Statements A and B could have been written in the form

```
F=F(M/N,N-1)*FLOAT(M)+F(M-N,N)*FLOAT(N)
```

However, it would then have been necessary to preserve the hidden location used to hold the first term while the second way is being calculated. As the location is hidden, this is impossible, so the split shown is needed to make TEMP explicit. Secondly, we have arranged to have only one RETURN in the routine for later simplicity.

We now define an inner section of F which operates on the Pth element of three arrays, M, N and TEMP, and which returns its results in TEMP(P). (A further array to hold the answer would be just a waste of space.) We also need a fourth array IEXIT, such that IEXIT(P) is to contain a value 1, 2, 3, 4 to indicate which exit mentioned above is to be used by the RETURN.

We get thus:

```
REC FUNCTION F(M,N)    FUNCTION F (M1,N1)
                       DIMENSION M(100),N(100)
                       DIMENSION TEMP (100),IEXIT(100)
                       INTEGER P
                       P = 0
                       M(1)=M1
                       N(1)=N1
                       IEXIT(1)=1
```

(We have now set up our stack for entry to the inner section which we will label 100. We also use labels 101, 102, 103 and 104 for the exits (1) to (4) mentioned earlier.)

```
                          100   P=P+1
                                IF(P.EQ.100) STOP 0123
      IF(N.EQ.0) GOTO 1         IF(N(P).EQ.0) GOTO 1
      IF(M.LT.N) GOTO 2         IF(M(P).LT.N(P))  GOTO 2
      TEMP=F(M/N,N-1)*FLOAT(M)  M(P+1)=M(P)/N(P)
                                N(P+1)=N(P)-1
                                IEXIT(P+1)=2
                                GOTO 100
                          102   TEMP(P)=TEMP(P+1)*FLOAT(M(P))
      F=TEMP+F(M-N,N)*FLOAT(N)  M(P+1)=M(P)-N(P)
                                N(P+1)=N(P)
                                IEXIT(P+1)=3
                                GOTO 100
                          103   TEMP(P)=TEMP(P)
                               1 +TEMP(P+1)*FLOAT(N(P))
      GOTO 3                    GOTO 3
    1 F=M                     1 TEMP(P)=M(P)
      GOTO 3                    GOTO 3
    2 F=F(N,M)                2 M(P+1)=N(P)
                                N(P+1)=M(P)
                                IEXIT(P+1)=4
                                GOTO 100
                          104   TEMP(P)=TEMP(P+1)
    3 RETURN                  3 P=P-1
                                IF (P.LT.0)STOP 4567
                                IGOTO=IEXIT(P+1)
                                GOTO(101,102,103,104),IGOTO
                          101   F=TEMP(1)
                                RETURN
      END                       END
```

The purpose of STOP 0123 is to detect stack *overflow* —too great a depth of recursion is needed for available store. The purpose of STOP 4567 is to detect stack *underflow* —this can only occur by an error in the coding.

The above example should give a clear idea of the technique. It is believed to be Standard FORTRAN. However, note some possible traps.

(a) To allow stacking we have to use a computed GOTO, and not ASSIGN and an assigned GOTO.

(b) We cannot recurse in the middle of a DO loop, since we are unable to preserve the hidden registers of the DO. In terms of the FORTRAN program, we are executing a DO within the extended range of itself, and hence attempting to redefine the control variable of a DO within the extended range of that DO. This is well known to be illegal. If a DO loop is needed, then it should be programmed using GOTOs, and a stack will probably be needed for the control variable.

DTIC COULD NOT GET THE

FOLLOWING MISSING PAGES

FROM CONTRIBUTOR

55

56

```
                                    GOTO 102
RETURN                      P = P - 1
                            IF (P.LT.0) STOP 4567
                            LINK = IEXITF(P+1)
                            GOTO 1000

                        200 LINK = 7

                       1000 IF (LINK.GT.6) CALL G
                            GOTO (101,102,103,104,105,106),LINK

                        101 F = RESF(1)
                            RETURN

END                         END

REC FUNCTION G(I)           SUBROUTINE G
                            COMMON/FANDG/M(100),
                           1   N(100),RESF(100),
                           1   IEXITF(100),I(100),
                           1   RESG(100),IEXITG(100),
                           1   P,Q,LINK
                             INTEGER P, Q, LINK
                             GOTO 1000

                        107 Q = Q + 1
                            IF (Q.GE.100) STOP 0123

G = G(...)                  I(Q+1) = ...
                            IEXITG(Q+1) = 8
                            GOTO 107                To 'CALL G'
                        108 RESG(Q) = RESG(Q+1)

G = F(...,...)              M(P+1) = ...
                            N(P+1) = ...
                            IEXITF(P+1) = 9
                            GOTO 200                To 'CALL F'
                        109 RESG(Q) = RESF(P+1)

G = G + F(...,...)          Similarly, label 110,
                                       IEXITF(P+1) = 10,
                                       GOTO 200

G = G + G(...)              Similarly, label 111,
                                       IEXITG(Q+1) = 11,
                                       GOTO 107

RETURN                      Q = Q - 1
                            IF (Q.LE.0) STOP 4567
                            LINK = IEXITG(Q+1)
                            GOTO 1000

                        200 LINK = 2

                       1000 IF (LINK.LE.6) RETURN
                            LINK = LINK - 6
                            GOTO (107,108,109,110,111),LINK

END                         END
```

(Note that the COMMON FANDG need only be declared in F and G.)

A number of variations are possible. For example, if F and G have the same number and type of arguments, the same arrays can be used; similarly, the arrays IEXITF and IEXITG, the two results arrays, and P and Q, can be merged. (This may even be sensible when the number of arguments differs.) A more elaborate store allocation scheme is also possible which wastes no space, but requires use of EQUIVALENCE (see later).

It should be noted that this same technique extends (with some slight loss of efficiency) to the case of several mutually recursive routines, any one of which may be entered first, and which may call each other in any order. In this case, an auxiliary routine (called RECURS, say) is desirable. Each routine has its own arrays for arguments, results, and local variables exactly as before. LINK exists as before, and each routine has a set of values (stored in the IEXITs) which LINK can take—one for 'entry', and one for the return from each (textual) recursive call. It would probably be sensible to put the RES, IEXIT, argument and local arrays for each routine together in a separate labelled COMMON, one for each routine. This would need to be declared in the 'owning' routine, in any routine calling (directly) the 'owning' routine, and in RECURS (to prevent undefinition on a 'CALL').

Each routine contains at its head

```
GOTO (—,—,—,—),LINK
```

and an equivalent of the sequence at label 1000 in subroutine G above.

The routine RECURS will identify the range of LINK, and then CALL the appropriate routine. The first call from outside the group of recursive routines (perhaps from the main program) is to RECURS, with LINK set to indicate which routine is being first entered.

If economy of store is important in the case of several mutually recursive routines, then a single region of store (perhaps open-ended in blank COMMON) can be used. To work in this way, suppose F requires RESF, IEXITF, M, N, and a local array WORK, and suppose G requires RESG, IEXITG, I, and a local array also called WORK. Then RECURS simply contains

```
COMMON//W(1000) say
```

and F contains

```
COMMON//W(1000)
DIMENSION RESF(1000),IEXIT(1000),M(1000),N(1000),WORK(1000)
EQUIVALENCE (W(5),RESF(1),IXITF(2),M(3),N(4),WORK(5))
```

G contains similar declarations. If F requires to call G, F will also need the DIMENSION and EQUIVALENCE statements for G's basic arrays, (but not for its 'locals') namely:

```
DIMENSION RESG(1000),IEXITG(1000),I(1000)
EQUIVALENCE (W(4),RESG(1),IEXIT(2),I(3))
```

The code for F then runs as follows:

```
F = F(...,...)                    M(P+5) = ...
                                  N(P+5) = ...
                                  IEXITF(P+5) = 3
                                  P = P + 5
                                  GOTO 102
                            103   RESF(P) = RESF(P+5)
```

where '5' is the number of storage units required by F's arrays. The increment at label 102 is of course omitted.

The call of G in F is:

```
F = G(...)                        I(P+5) = ...
                                  IEXITG(P+5) = 4
                                  P = P + 5
                                  GOTO 200
                            104   RESF(P) = RESG(P+5)
```

and so on. The increments to P in G will be by 4 (in our example) no matter what it calls. Decrements are characteristic of the routine being 'RETURN'ed to, and should therefore be done there.

This completes the discussion of recursive programming. Preprocessors do exist to implement schemes of this general nature, but provided the mechanisms are understood, it is usually not too tedious to do the coding by hand, and usually much more efficient!

3. OPTIMIZATION

There are a number of aspects to efficiency, the most important usually being the choice of the right global strategy. There is little that can be taught about that here, since it involves algorithm design, a topic beyond the intended scope (see [9] for examples). An approach at the detail level will try to discover which part of a program is taking up the majority of the time: it will usually be found that less than ten per cent of the written code accounts for over fifty per cent of the time the program is taking. In some cases, it is possible to identify by a mathematical analysis the number of times various parts of the program are being obeyed. For example, if a program is of the form

```
      ...
      ...
      ...                 A
      ...
      ...
      DO 1    I=1,N
      ...
      ...
      ...                 B
      ...
      ...
1
```

```
   ...
   ...
   ...                    A
   ...
   DO 2    I=1,N
   ...
   ...
   ...                    B
   DO 3    J=I,N
   ...
   ...
   ...                    C
   ...
3
   ...
   ...
   ...                    B
   ...
2
   ...
   ...                    A
   ...
   ...                        (N constant)
```

then, giving equal weight w to each statement, and assuming *a* statements altogether in sections A, and *b* statements in sections B, etc., we have a time of

$$(a + b*N + (1/2)*c*N*(N + 1))*w$$

For sufficiently large N, only section C matters. For small N it may be useful to look at section B as well.

If, as often happens, the above analysis is not possible, it is necessary to insert counting statements, thus

```
N(1) = N(1) + 1
 ...
 ...
 ...
 ...
N(2) = N(2) + 1
 ...
 ...
 ...
 ...
 ...
N(3) = N(3) + 1
etc.
```

at suitable points in the program. (This can be done automatically, as described in other

sections.) It is necessary to provide a sufficiently large amount of data to exercise the program in the way it is likely to be used, and for any sizeable program a routine will be needed to summarize the N array at the end. If done properly, the increment will not be by one, but by the number of statements in that section, and the final summarizing routine will be able to say precisely which parts of the program are needed to account for various percentages of the time.

It is of course important not to forget that FORTRAN statements do not all take an equal time, and a system facility which counts machine instructions is preferable, but this is usually more expensive to provide, and it is more difficult to produce its output in terms of the original source. However, a weight factor of 50 to 100 for a CALL statement or a function call, on top of the instructions obeyed in the subprogram, will occur in some implementations. Few installations will do better than 5 to 10.

Once the important, time-consuming part of a program has been found, it may indicate a total change in the algorithm. If not, the problem is simply one of coding that section efficiently. We now therefore need to consider local optimization, which is an area which can be handled by an optimizing compiler. (Usually, of course, an optimizing compiler is used for the whole program, and so will 'waste' time optimizing completely non-critical parts. We mention the problem of minimizing the written length of a program—that is, the amount of store it takes. This is an area in which every line of the program counts, and not just inner loops, so an optimizing compiler is a major asset.)

However, not all machines have optimizing compilers available, and the user can do much by being aware of the code likely to be generated by his statements. The task of changing

```
    IF (I - 0) 22,1,2
 2  IF (I - 1) 4,3,4
 4  IF (I - 2) 6,5,6
 6  IF (I - 3) 8,7,8
    etc. up to
20  IF (I - 10) 22,21,22

    into

    IF (I) 22,1,2
 2  IF (I - 10) 4,21,22
 4  GOTO(3,5,7,...,19),I
```

is one which few (if any) optimizing compilers will manage.

It is in fact possible that the variable I can never be negative or greater than 10, so that

```
I = I + 1
GOTO(1,3,5,7...,19,21),I
```

is correct code. No compiler can be expected to know that the tests included in the original version are not needed. (If I had been a REAL, the corresponding sequences could have been better one way on time, and better the other way on space.)

Now let us get down to specific cases. Suppose we have identified the following sequence as our inner loop:

```
      DO 1 I = 1,N
      J=3*K+I
    1 B(J)=A(J)+1./C+FLOAT(I)/10.
```

what can we do with it? First of all, 1./C is putting in a division every time round the loop. Similarly, 3*K is being calculated unnecessarily. (In many systems, the non-standard expression

```
B(3*K + I) = A(3*K + I)...
```

would have been accepted, and would have been even worse.) Let us write

```
      C1 = 1./C
      J1 = 3*K
      DO 1 I = 1,N
      J = J1 + I
    1 B(J) = A(J) + C1 + FLOAT(I)/10.
```

Note that the removal of common sub-expressions from a group of two or three statements is also worthwhile.

```
A(1) = (X - Y)*(X + Y)
A(2) = (X - Y)*(X + 2.*Y)
```

can be written

```
A1 = X - Y
A(1) = A1*(X + Y)
A(2) = A1*(X + 2.*Y)
```

or even better

```
A1 = X - Y
A(1) = A1*(X + Y)
A(2) = A(1) + A1*Y
```

(Assuming that there are not numerical analysis problems preventing these changes.)

But let us return to our DO loop. We now have

```
      DO 1 I = 1,N
      J = J1 + I
    1 B(J) = A(J) + C1 + FLOAT(I)/10.
```

Now on most machines the FLOAT and FIX operations are fairly expensive, up to five times a normal add or subtract, perhaps. So we might consider

```
      C2 = 0.
      DO 1 I = 1,N
      C2 = C2 + 0.1
      J = J1 + I
    1 B(J) = A(J) + C1 + C2
```

or better

```
      C2 = C1
      DO 1 I = 1,N
      C2 = C2 + 0.1
      J = J1 + I
    1 B(J) = A(J) + C2
```

However, here we must be careful since 0.1 will not be held exactly, and there will be a large error buildup in C2 if N is large, whereas FLOAT(I)/10. always gives maximum accuracy.

A reasonable compromise between accuracy and efficiency might be

```
      C3 = C1*10.
      DO 1 I = 1,N
      C3 = C3 + 1.
      J = J1 + I
    1 B(J) = A(J) + 0.1*C3
```

(There is no error build-up in C3 because 1 to N, being fairly small integers, will be held exactly. C3 is simply a scaled up version of C2.)

We can now get rid of J.

```
      J2 = J1 + 1
      J3 = J1 + N
      DO 1 I = J2,J3
      C3 = C3 +1.
    1 B(I) = A(I) + 0.1*C3
```

So the final inner loop becomes

```
      DO 1 I = J2,J3
      C3 = C3 + 1.
    1 B(I) = A(I) + 0.1*C3
```

This looks like fairly tight code, but we can in fact do still better if we are prepared to go to a lot of trouble. We first illustrate the technique—DO loop expansion—on a very simple case, then return to our actual loop.

Let us suppose that A(I) = 0. compiles into two instructions, and that a DO loop repeat compiles into three. (The figures are fairly realistic, but it would not matter if they were 1 and 1 or 3 and 1, the general principle still holds.)

Then

```
      DO 1 I = 1,160
1     A(I) = 0.
```

takes

```
      160*(2 + 3) = 800 instructions,
```

```
      DO 1 I = 1,159,2
      A(I) = 0.
1     A(I + 1) = 0.
```

takes

```
      80*(2 + 2 + 3) = 560 instructions,
```

and

```
      DO 1 I = 1,157,4
      A(I) = 0.
      A(I + 1) = 0.
      A(I + 2) = 0.
1     A(I + 3) = 0.
```

takes

```
      40*(2 + 2 + 2 + 2 + 3) = 440 instructions.
```

Thus, by writing out the interior of the loop several times, we can spread the overhead of the loop repetition broadly enough that the cost for each item is effectively zero. Note that the technique only gives worthwhile gains if the number of instructions obeyed within the loop is comparable to the number of instructions taken for the repeat.

Now we will return to the DO loop we are trying to optimize. Let us suppose we know N near the head of the program. Then we form, say

```
      N1 = N/8
      N2 = N - 8*N1
```

The figure 8 is an arbitrary one. Assume N1 $>$ 1, and N2 $>$ 1 (some simple additional tests sort things out if they are not).

The original loop:

```
      DO 1 I = 1,N
      J = 3*K + I
1     B(J) = A(J) + 1./C + FLOAT(I)/10.
```

is then equivalent to:

```
      DO 1 I1 = 1,N1
      DO 1 I2 = 1,8
      I = 8*(I1 - 1) + I2
      C3 = C3 + 1.
      J = J1 + I
1     B(J) = A(J) + 0.1*C3
      J4 = 8*N1 + 1
      DO 3 I = J4,N
      C3 = C3 + 1.
      J = J1 + I
3     B(J) = A(J) + 0.1*C3
```

The inner loop can now be written:

```
      K1 = 8*(I1 - 1) + J1
      DO 1 I2 = 1,8
      C3 = C3 +1.
      J = K1 + I2
1     B(J) = A(J) + 0.1*C3
```

and so we expand it into its equivalent sixteen statements:

```
      C3 = C3 + 1.
      B(K1 + 1) = A(K1 + 1) + 0.1*C3
      C3 = C3 + 1.
      B(K1 + 2) = A(K1 + 2) + 0.1*C3
      ...
      ...
      ...
      C3 = C3 + 1.
      B(K1 + 8) = A(K1 + 8) + 0.1*C3
```

We can in fact do better still by accepting the error build-up in 0.1*C3 by adding up to eight times, then recalculating the product every time:

```
      C4 = (C3 + 1.)*0.1
      C3 = C3 + 8.
      B(K1 + 1) = A(K1 + 1) + C4
      C4 = C4 + 0.1
      B(K1 + 2) = A(K1 + 2) + C4
      etc.
```

Thus in our original loop, for each element of the arrays we obey

```
      J = 3*K + I
      B(J) = A(J) + 1./C + FLOAT(I)/10.
      {repeat}
```

Suppose three instructions for the repeat, and five instructions for the FLOAT, then

```
      J = 3*K + I is, say, four instructions
      B(J) = etc. is, say, eight instructions
```

giving a total of twenty instructions.

In our final version, we have only

```
C4 = C4 + 0.1
B(K1 + n) = A(K1 + n) + C4
```

Say three instructions for the first statement and three for the second, giving a total of six instructions.

This section of the program will thus run over three times as fast as it did originally for N sufficiently large (or almost—we have neglected one-eighth of the overhead at the end of each group of eight statements).

However the final version of the loop will take up *much* more core, and is almost impossible to understand! One should also note that even if the original loop accounted for 60 percent of the complete program's time, the total time taken is still 20 + 40 per cent—i.e., we have not quite halved the total time. So the techniques in their extreme form should only be used on inner loops, and the original intelligible version should be provided as comment. Some of the early changes made, however, can be widely applied.

It is left as an exercise to the reader to write out the complete program which replaces the original loop, and to calculate exactly (given reasonable assumptions for the number of instructions produced by each statement), the number of instructions obeyed in each case as a function of N. How large must N be to break even? What is the ratio of written instructions in the modified program to the original one (i.e. roughly, the increase in size)?

4. FORTRAN INPUT/OUTPUT

The input/output facilities in FORTRAN are at first sight very flexible, but you will soon find inadequacies and problems when you start to do real work. In general, you will find two solutions to your problems. The first is to use local non-Standard facilities, designed to cope with your difficulty in a simple and efficient manner. The second is to write some fairly complex routines, involving a degree of character handling, using the more elaborate features of the Standard, and doing some of the binary-decimal conversion normally handled by the package. There is in fact a third solution, frequently adopted, namely to accept an inferior interface to the user, so that the simple Standard features will be adequate. The examples which follow will make these choices clearer.

Let us concentrate first on output, as this is perhaps the simpler problem. Straightforward use of the system is possible provided the following conditions are fulfilled:

(a) Your program is structured in such a way that complete lines of output are produced before writing the first number.

(b) You know when you write the program the rough sizes of the resulting numbers, and can therefore choose the most appropriate format.

(c) The number of items to be put on each line. (indeed the whole line format) is known when you write the program and does not depend on the data.
(d) The "space" character is acceptable as a background character for filling out small numbers to the appropriate field width.

Taking these points in turn, we look at the output of part lines. Use of 1H+ as the carriage control character does allow a line to be printed in several parts *of known size.* A different FORMAT statement would be needed for each part, so a count must be kept. However, although 1H+ *is* available in the Standard, it is inadequately defined, and there may be problems with some systems. It is usually far preferable (and more efficient) to buffer a complete line yourself.

The sizes of numbers is a difficult problem. Consider for example the output of

```
129 27.65 0.03 3.276E4
```

Suppose you would like numbers like these printing in these styles, but you don't know which of your results will fall into which category. In the simple system you cannot do it, but must choose in advance one of the available fixed formats. If you use multiple WRITE statements with 1H+ as described above for part lines, then for each number you could select one of several FORMAT statements, to provide the clearest output. (But even this is not easy, for each WRITE statement has a fixed FORMAT statement, so you need to use branching to select different WRITE statements.) The solution suggested below for the next problem is also available—constructing a FORMAT statement, and for anyone wanting very pretty output, this is almost the only answer.

Output requiring an unknown number of items per line occurs most often in the production of graphs or histograms on the line-printer. In the simplest case it is required to space across the page an amount depending on the data, then to print an asterisk. This requires a *variable* count in the FORMAT statement, and cannot be simply done. The easiest solution is to generate an array of 120 (say) characters (in A1 format), all of which are spaces except for one asterisk in the appropriate place. Printing the array in 120A1 format solves the problem. However, the construction of the array, and the type of variable holding it, requires some care. As was pointed out in the earlier chapter "Standard FORTRAN—The Rules of the Game", character handling in FORTRAN is full of pitfalls. Copying REAL variables holding characters often fails, and comparison of INTEGER variables holding characters sometimes fails. (More often with IF(I-J) than with IF(I.EQ.J).) In this case we only require *copying,* so use of INTEGER variables would be safest. The code to set up the array should, however, be carefully isolated and noted in case it needs changing on another system. The "space" and the "asterisk" could be set up in a DATA statement within a subroutine which is passed N—the position for the asterisk, and IOUT—the array to hold spaces and the asterisk.

This technique is not always adequate. Suppose we also wish to follow the asterisk with a printout of the value it represents, not at the right of the paper, but immediately following the asterisk. We are now forced to exploit an underused facility provided by the Standard, but which *does* seem to be provided by most systems. This is *constructing* your FORMAT statement at run-time. Spaces may appear anywhere within a FORMAT statement, and if you construct

your FORMAT in an array using A1 format, you will have a number of spaces between each character, that number being implementation dependent, but this will not matter. As before, the characters to be used will be set up in a DATA statement, and the *copying* of characters which is needed should be carefully isolated. Using this technique the power of the FORTRAN output facilities is greatly extended, and any reasonable output problem can now be solved.

The final output problem to be addressed is that of changing the "background" character. (Bills and checks, for example, often have the field filled with asterisks.) Very efficient, very simple, local facilities are usually provided, to simply "tell" the I/O package to use asterisk (say) instead of space. (Comma for decimal point for output intended for continental Europe is another example.) Remaining in Standard FORTRAN, however, we have two choices. We can test the magnitude of our numbers and then select or construct a suitable FORMAT. However, very often a package will insist on room in a field for a sign character, and then put a space in the plus case. Constructing a FORMAT cannot overcome this problem, and we are driven to the final step—writing a routine to convert a numerical value into decimal, and hence into characters, and then printing the entire line in A1 FORMAT. This is a lot of work, will be much less efficient than direct use of the package, and is only rarely justified. It is, however, the complete answer to "you cannot do it in FORTRAN" where output is concerned.

If we turn now to input, we again find complete flexibility, provided we can read a complete line at a time, and provided the layout of each line to be input is known when the program is written. Otherwise we have five main problems to be addressed: terminating the data (or part of it); reading items of no fixed length, terminated by space (or newline or comma); reading items which may be either numerical or alphabetical codings; producing sensible errors for the user if he provides bad data; and finally, general problems with end of line.

Terminating the data is most naturally done by hitting 'End of file,' or by hitting some nondigit (like /). Unfortunately detecting 'End of file' is not possible in Standard FORTRAN, nor is trapping of number conversion errors. Provision of the first of these (albeit with widely varying syntax) is very widespread, the trapping less so. As always, a decision to use the non-Standard 'End of file' facility should be consciously taken, and the code carefully isolated. (For example, provide a routine which will read the data and buffer it, returning one item on each call of the routine, together with a Boolean set .TRUE., and setting the Boolean false when no more items are available. There are other ways of communicating—see Section 5.

If the non-Standard solution is *not* acceptable, then the data must be terminated by a numerical value or sequence of values which cannot occur in the data (any randomly chosen set of six six-digit integers or reals will be *very* unlikely to appear given 'random' data—but many people like certainties), *or* the number of items must appear as the first item read. This latter approach is inconvenient for the user of the package, but does allow a measure of checking that the data is complete.

Our next problem, usually termed free-format input, is reading items of unknown size (into REALs say), terminated by spaces or newline. Once again, local facilities are quite widespread, but in Standard FORTRAN there is no solution to this problem without going to the extreme of writing a routine to recognize characters (comparison of input character— A1— with DATA statement) and produce a numeric coding for each character.

The binary value can then be built up. This will usually produce a fairly inefficient input system, but may be acceptable for small quantities of data. The best approach is to provide a routine which reads the data, and converts each character to a numerical coding fully documented. This routine should probably operate by reading A1 format into INTEGER's and comparing in turn with each of the possible characters set up in the DATA statement. Documentation of the package should mention the routine, and in particular the potential need to change to use of REAL's on some machines. The routine could be replaced locally by one which used a local facility to give a local numeric coding, and an ad hoc program to convert this into the required coding.

We now turn to the problem of attempting to read data which may consist of numerical or alphabetical quantities. This sort of problem usually arises when the data exists before the program is written, and the simplest solution is often to pre-process the data using A-format, or a non-FORTRAN program such as an editor. It can, however, arise where the requirements of a neat user interface require lines of text as well as lines of numbers to be processed. (Perhaps comment which is to be ignored, or subtitles to be reflected into the output.) What is needed is the ability to 'look' at a character, then to use the I/O package to read it in one of several ways. There is at least one package circulating which reads a line in A format, decides how it should be read, writes it out to a temporary unit, REWINDs that unit, and reads the line in the appropriate way. This is very cumbersome and inefficient, and moreover assumes that the system will provide temporary units which can be freely written, rewound, and read at little expense. Use of the Standard FORTRAN statement BACKSPACE is another possibility to achieve the desired effect, but this is unfortunately one of the more common omissions from the Standard, (at least for some input media), depending usually on the provision of an equivalent facility by the operating system.

The next problem to be addressed—coping with bad data—is a generalization of the previous problem, but is much more important. Any package intended to be used by non-computer people (and perhaps just any package!) should be robust. A collapse because a key puncher has mis-keyed the data is most unsatisfactory (but regrettably common!). Unfortunately there is very little which can be done in Standard FORTRAN, short of the technique mentioned earlier of reading all the data in A1 format and either doing the conversions by program or, slightly more dangerously, checking the syntax with A formats and producing diagnostics, then rewinding the data and reading it properly. There are a few local extensions available to help in this area, but they rarely provide the information needed for a good diagnostic and/or error recovery by the package, and there is nothing widely available. (It is of course important and possible for any program to verify that its input parameters are within any expected ranges, and to give an error if they are not, but this has nothing to do with FORTRAN input/output as such.)

Finally, a short discussion of the problem of 'End-of-line' is appropriate. FORTRAN grew up with input from punched cards. Thus every line was 80 characters long, and life was (fairly) simple. On paper-tape oriented machines, however, an explicit newline character (or combination of characters) can appear at any point, and lines are of arbitrary length. Furthermore, most operating systems at least *allow* lines to be of variable length for data from files, and terminal input has this natural form. Very often the FORTRAN system will insist on 80 character lines, and some very confusing situations can arise, not least being 'trailing spaces' problems. Detection of 'End of line' is not possible in Standard FORTRAN, and applications where this is important are forced to use local facilities.

In some cases input files can contain 'carriage control characters' more elaborate than the simple 'newline,' and appearing sometimes at the start of a line, sometimes at the end. The precise appearance of files going into FORTRAN is not defined by the Standard, it being properly the province of the operating system. The only advice to be given here is to try to use simple formats if possible. If a program must be written to accept and act sensibly on all input formats at an installation, that program will contain many uses of local facilities, and some care will be needed to properly isolate them.

In FORTRAN input/output programming more than any other, much effort and care is needed to achieve the right balance between a good interface to the program and the need to remain Standard. Keeping the body of the program Standard while using local facilities is also a skilled and difficult task. It can be achieved if care is taken.

5. USER INTERFACES

We begin with a discussion of the user interface and storage allocation of one particular fairly large package produced by the author.[3] The body of the package is concerned with performing integer arithmetic with arbitrary large numbers. There are several questions to be asked in designing such a package.

(a) How are we to obtain storage (memory) for holding the numbers? How will the user refer to some number?

(b) What operations must be provided in a general purpose package?

(c) What global switches are needed, how are they held and how does the user set them? How do we handle default settings?

One solution, widely adopted in similar packages, is to make the user hold each integer in a fixed length array declared in his main program. This has two advantages—the package need not waste time finding a variable or getting new memory, and secondly, the user has direct access to the elements of the array from his program, and hence to the number itself. This means that he himself can answer questions such as: Is the number zero, positive or negative? Does it exceed some value? He may also be able to negate it directly and to set an ordinary integer value into it himself. If he goes into a loop and the number grows very large, he is soon stopped by the length of his array (i.e., he probably overwrites the next array or program, or is caught by the package).

Disadvantages also exist as a direct consequence. He has to declare all his VPIs (variable precision integers) as arrays, and so each one is of fixed length, and storage must be allocated for the largest value any can grow to. If he wants a one- or two-dimensional array of VPIs, he must declare instead a two- or three-dimensional array, respectively, to hold them, and be careful to hand over the right subscript. In general, then, this is a user interface with some advantages, but definitely second rate.

We will now look at a use of the author's package. Let us suppose we want to use the package and that we will, in the main program, have I1, I2 and the array NUMB(20,2) as VPIs. We declare NUMB in the usual way

```
DIMENSION NUMB(20,2)
```

and then tell the package what space it can have for holding the VPIs by

```
CALL USE(1,1000)
```

This gives to the package the first thousand locations of blank common. Any other area of COMMON could have been used. This entry also enables some initialization to take place to set up default values in a piece of labeled COMMON called /VARP/ (this is better than making the user set them up), e.g. IBASE, the base of the number system to be used; ILEN, the maximum length of number to be allowed before giving an error, etc. The parameters of USE can also be preserved in this area.

There are two choices here. The documentation can describe this labeled common, so that if the user wishes to change some of these values he can simply declare the common, and write code to directly change the location. The second approach is to provide a routine or routines which will change one or more values in the common. If it is felt that the values are unlikely to be changed by many users, the former course may be taken, but the second is to be preferred in a workmanlike package. The amount of documentation needed is slightly less in the second case, but more importantly, if a future release of the package needs to change the layout of the labeled common, or the type of the value, then the end user does not need to make corresponding changes to his program. The disadvantages of the subroutine approach, particularly if one subroutine is provided for every parameter, is that it increases the number of global names in the package, and hence makes it more difficult to remember and learn to use.

Next the user inserts

```
CALL BEGIN
```

(The purpose of this will be seen later.) Then

```
CALL VAR2(I1,I2)
CALL ARR1(NUMB,40)
```

This is the first interesting point. The user is telling the package that these integer variables and the array are to be used for holding VPIs. The FORTRAN variables are given values (1, 2, 3, ...in fact!) which serve to identify, via a table, the place in the COMMON area where that VPI is stored. Note that we really require a routine VAR (and another ARR) which takes an arbitrary number of arguments. Unfortunately, this is not possible in FORTRAN, so we get the same effect by providing nine routines VAR1 to VAR9 taking one to nine arguments each. This is just as convenient as the user writing, e.g.

```
CALL VAR(3,I1,I2,I3)
```

(if he were allowed to) and only slightly less convenient than

```
CALL VAR(I1,I2,I3)
```

The opportunity is also taken to zero the VPI values of these variables. Thus the user can proceed with his calculation, and in due course may enter a subroutine. Integers declared as VPIs can be passed through COMMON or as arguments in the usual way, but the user may want some variables local to the subroutine as VPIs. In this case, he writes, e.g.

```
CALL BEGIN
CALL VAR3(J,K,L)
```

and does his calculations. Now we see the purpose of CALL BEGIN. Immediately before the RETURN, the user writes

```
CALL END
```

and the effect of this is to 'undeclare'—to get back for future allocation— all space used by any VPIs declared since the last occurrence of CALL BEGIN.

The variables J, K, L will then be redeclared, if necessary, on the next entry. Note that this serves two functions. Firstly, it ensures that we do not have space tied up by these local VPIs except when they are active—i.e. we are in the routine. Secondly, we do not require the FORTRAN integers to retain their values from the exit to the next entry—something the Standard prevents us from doing.

Now let us look at the calculations we can perform. The obvious first requirement is an input routine and an output routine for arbitrarily large integers. Here we come to an important principle. It is usually best to leave all actual input/output to the user himself. He can then use the full flexibility of FORTRAN to control his layout, intersperse text, handle numbers which spread over several lines, and so on. Thus what the user does is to read into a suitable sized array in A1 format the digits forming the number to be 'read'; this array, A say; its size, 80 say; a VPI, J say, to hold the result are then passed to the 'read' routine:

```
CALL IRD(J,A,80)
```

Similarly 'printing' is into an array, and the user will do the actual output with A1 format. He also needs a function

```
N = LEN(J)
```

which will return the number of places needed for the printing of the VPI J.

Now we should consider the main arithmetic routines. There are four possible user interfaces, exemplified by:

(i) CALL ADD(I1,I2,I3)
sets I1 to I2 + I3
(called a three-address system).

(ii) CALL SET(I1,I2)
sets I1 to I2

CALL ADD(I1,I3)
adds I3 to I1
(called a two-address system).

(iii) CALL LOAD (I2)
puts I2 in a hidden accumulator;

CALL ADD(I3)
adds I3 into the accumulator;

CALL STORE (I1)
stores the accumulator in the location I1
(called a one-address system).

(iv) CALL LOAD (I2)
put I2 onto the top of a stack

CALL LOAD (I3)
put I3 onto the top of the stack

CALL ADD
adds the top two items,
leaving the result on the stack

CALL STORE (I1)
stores the top item in I1
(called a stack system, sometimes zero-address,
which is true for the operations, but of course
not for LOAD and STORE).

These four sequences (i)—(iv) all perform

```
I1 = I2 + I3
```

Now although (i) looks the most convenient for a single statement, it is more expensive than (ii) when a sum is to be accumulated. Compare the copying of VPIs involved in

```
  CALL VAR1(I)
  CALL ARR1(A,N)
  ...
  ...
  ...
  DO 1 J = 1,N
1 CALL ADD(I,I,A(J))
  with
  DO 1 J = 1,N
1 CALL ADD(I,A(J))
```

The extra flexibility of the two-address system can be useful in some case, and there is no further gain in going to one-address. The stack system can be useful in minimizing work where location of an item is difficult or expensive, but is generally more cumbersome for the user. (Its logical simplicity often appeals to computer scientists however, and is the basis for one major line of hand calculators.)

The differences between these various schemes are fairly marginal, and in this case we adopt a two-address approach.

Just to give the reader a feel for how much needs to be provided in addition to ADD, SUBTRACT, MULTIPLY and DIVIDE, we briefly list the routines available in the package. We use names beginning with A or F for reals, D for doubles, N for ordinary integers, I for VPIs.

Firstly, there is a batch of routines for copying values from every sort of variable to every other sort—the equivalents of the ordinary FORTRAN A = I and I = A.

```
NSET(I,N)          I = N
ASET(I,A)          I = AINT(A)
DSET(I,D)          I = DINT(D)
SET(I1,I2)         I1 = I2
EXCH(I1,I2)        I1 = I2;I2 = I1
```

('EXCH' is available because I1 and I2 merely contain pointers to the actual VPI; it is very cheap to simply switch pointers. Doing the operations of EXCH with SETs would be much more expensive.)

```
FLT(I)             FLT = I
DLFT(I)            DFLT = I
```

The actual arithmetic routines are obvious.

```
ADD(I1,I2)         I1 = I1 + I2
INCR(I)            I = I + 1
```

('INCR' is provided because adding one is so common. Note that we could instead have provided 'mixed-mode' expressions with

```
SADD(I,N) I = I + N
```

but this was thought unnecessary.)

```
SUBT(I1,I2)        I1 = I1 - I2
DECR(I)            I = I - 1
SMULT(I,N)         I = I*N
MULT(I1,I2)        I1 = I1*I2
SDIV(I,N1,N2)      I = Quot(I/N1),N2 = remainder
DIV(I1,I2,I3,I4)   I3 = Quot(I1/I2),I4 = remainder
```

(Note that in this case we have gone to three-address form.)

```
NEG(I)             I = - I
ABSOL(I)           I = |I|
```

The next group of routines we need enables us to branch on the state of a VPI by using, e.g.

```
IF(ISIG(I))---,---,---
```

Thus

```
ISIG(I)          +1 if I>0; 0 if I = 0; -1 if I<0
ICOMP(I1,I2)     +1 if I1>I2; 0 if I1 = I2; -1 if I1<I2
NCOMP(I1,N)      +1 if I1>N; 0 if I1 = N; -1 if I1<N
```

Finally, we have some 'external functions' as it were, which are written using the basic routines

```
GCD(I1,I2,I3)        I3 = Gcd(I1,I2)
ISQRT(I1, I2,I3)     I2 = Sqrt(I1);I3 = remainder
POWER(I1,I2,N)       I1 = I2**N
MPOWER(I1,I2,I3,I4)  I1 = (I2**I3) mod I4
NMOD(I,N)            I = I mod N
IMOD(I1,I2)          I1 = I1 mod I2
```

The purpose of the above is not to teach the use of a VPI package, but to give a feel for the range of facilities which must be provided in that sort of package. Note in particular that if the VPI had been held in a fixed length array provided by the user, the routines NSET, NEG, ABSOL, ISIG would probably not have been needed—the user could have done them by direct operation on the VPI. However, the saving in work would not have been balanced by the disadvantages.

The next point to be covered is the handling of errors. These will be detected at a great many points—'number too large for available storage,' 'attempt to divide by zero', 'ordinary integer used where VPI expected', etc. They are best handled by arranging to call a single error routine, with the error number, to put out a diagnostic message. If a return occurs, the routine should tidy up as well as possible, and itself return. Normally the error routine will stop after the diagnostic (an exception might be 'non-digit in number' in IRD), but by having a single routine the user can, if he wishes, modify or replace it to take some special action. Any WRITE statements used for diagnostics should have the unit number held in a variable which is set in the initialization sequence, so that its value can be changed.

Finally, we mention briefly the storage allocation for the VPIs—i.e. how the package works. Each variable, I1, etc., used by the user points, not directly to the VPI, but to a stack, which then points to the VPI. (A VPI value of zero has zero in the stack to save space, and to avoid the problems of having a positive and negative zero.) Whenever we declare a VPI a new entry is made on the stack. On a CALL BEGIN, we simply place a unique value, a bar on to the stack. On a CALL END, we free all storage pointed to by any stack entries beyond the last bar, and clear the stack back to and including the bar. The point about having this indirect reference is to enable VPIs (held as contiguous blocks of storage in the work area) to be moved about if the store gets too fragmented.

There are many possible techniques for handling store in this sort of way, but this is a specialist field and beyond this text. The key words are 'garbage collection, space allocation, storage compaction'. For further details consult [4].

Now let us consider a somewhat different problem. We have to produce a local routine to sort the rows and columns of a matrix (real or integer) into order, by some row or column. The first thing we observe is that the user will probably not have filled the whole matrix, or will only want to sort on part of it. The next thing we note is that he may want to have some column (or row) in ascending or descending order. How do we define the user interface to such a routine? (Before continuing, the reader may like to invent his own interface.)

One possibility is to code the options:

0 means rows ascending
1 means rows descending
2 means columns ascending
3 means columns descending

Another approach is to have more arguments to keep it simple and use 0,0; 0,1; 1,0; 1,1. If we were restricted to Standard FORTRAN we might well have to put up with such an interface—completely impossible to remember. However, for a local routine, we will be able, on most systems, to let the user write

```
CALL SORT('REAL',A,M,N,'COL',3,'ASCENDING',M1,N1)
where  'REAL' could be 'INT'
       'COL' could be 'ROW'

and    'ASCENDING' could be 'DESCENDING'
```

and the example above means

Sort the **real** *array A of size (M,N) so that column 3 is in* **ascending** *order, sorting only the 1 — M1, 1 — N1 part of the array.*

If we include a few synonyms like

```
COL = COLS = COLUMN = COLUMNS etc.
```

then we have a very reasonable user interface which can be very easily remembered. Perhaps the final bit of sugar would be, if possible, to allow M1 and N1 to be omitted if they equal M and N. (Note that what the user is writing is Standard FORTRAN. Deviations appear only when we try to write the routine being called! This non-standardness can be kept to the first few statements which decode the arguments.)

One change can be accepted and makes it more nearly possible to cope, and that is to absorb the REAL or INT into the name:

```
CALL REALST (A,——)
CALL INSTST (I,——)
```

We can then obey the Standard's directive to match the types of the arguments; the Hollerith string handling (at least the first four characters, say) will go through on most installations, and the routine is much more transportable than at first it appeared.

These two 'case studies' have brought out some of the points to be considered in the design of interfaces to the end-user. We have seen examples of the various ways of passing arguments—in the call, in an accessible piece of labeled COMMON, or by a preceding routine call to set a global switch in a hidden labeled common block. The most appropriate mechanism depends on the frequency of changes between different textual calls, and on the size and complexity of the system being written.

We have also touched on the problem of error handling. There are some programmers who would argue that all SUBROUTINES should be made into LOGICAL functions, and called by

```
IF (SUBR(.....))GOTO 100
```

instead of

```
CALL SUBR (.....)
```

where 100 is the address of the error exit. There are others who feel that errors should call a global routine and stop. In general, any package must be written with careful thought to action in error cases. Lack of decision on how to deal with an error often leads to a failure to test for errors, or inappropriate action on error cases. FORTRAN is not well suited for handling errors (indeed, there is no consensus on what facilities a language should provide in this area), and each package must develop its own conventions.

The message of this section is that considerable thought should be given to any user interfaces to make them 'clean' and as easily recalled as possible, and to avoid passing unnecessary information or causing repetition. (Remember the use of COMMON, possibly labeled, for holding global switches and passing values which change rarely.) Try to achieve the right compromise between slavish adherence to the Standard and going overboard to get a neat interface.

Where dynamic space allocation is a major asset to the package, do not be afraid to program it yourself when it is not available in the language you are using.

6. STRUCTURING YOUR PROGRAM

'Structured programming' is today a familiar phrase, and is usually associated with comments like 'don't use gotos' and with FORTRAN extensions involving IF ... THEN ... ELSE and REPEAT ... UNTIL statements. The purpose of this section is, however, to take a wider view of program structure, and to indicate some of the structuring devices available in Standard FORTRAN.

The purpose of structuring a piece of program is to make it easier to write, debug, modify, and for somebody else to read. The ways this is achieved are as follows:

(1) For each logical hunk of data (perhaps two arrays and a pointer) arrange for accesses to it to be localized in the code.

This means that if a change is made to the layout of this data, a relatively small piece of program has to be examined. It also makes it relatively easy to check and to debug the code modifying this data, establishing invariant conditions which always hold if it is correct.

The easiest way to achieve this objective in FORTRAN is to put the data in its own labeled common block, and to declare that block only in the routine or routines which use it. (Other parts of the program will access it through those routines—compare the remarks on changing default settings in the VPI package.)

(2) Try to ensure that a single flow of control appears frequently in the program—to elaborate, if vertical lines are drawn from each GOTO or arithmetic IF to the corresponding label, and if horizontal lines are drawn through statements which are not crossed by a vertical line, then you should try to ensure that there is never more than perhaps a page of program between each horizontal line.

This means that a human reader can follow through the piece of code and hold in his head the various possible ways of branching, and in general what the program is doing. If comments are inserted where each of the horizontal lines above appear, describing what the following block is doing, then this helps a great deal.

This aim can be achieved partly by a careful study before coding of the sort of program structure most suited to the problem. It can also be aided by removing large hunks of code into a subroutine (called only once). This is naturally achieved if a 'top–down' program development is used, where the program is written assuming the existence of certain routines which are not themselves written until the next stage of the iteration. [I do not wish to press any form of program development— I favor top–down design and bottom–up coding, but everyone has his own personal approach; the end result is what matters!]

(3) Group logically related storage areas together and use any available linguistic device to show the grouping.

This is designed to aid readability. It covers such ideas as declaring related variables in one INTEGER or DIMENSION declaration, and another group of related variables in *another* declaration on the next line. (Again, a comment saying what the group is, is always worthwhile.) It also covers use of names which have some structure—the first few characters being the same for related variables—stated conventions on the use of initial letters, and so on. Finally, there is the already mentioned technique of putting related variables in a labeled common block—even in a program with no subroutines or functions.

(4) Group logically related functions and subroutines together.

There are those who favor listing subroutines *before* the code which uses them, others who prefer them *after*. Once your program is complete, consider whether it would be more understandable if the order of your functions and subroutines was changed. Review the names you have used, and if necessary, change them to reflect the relation between procedures, or the level at which they are called.

(5) Try to keep the number of different effects of a piece of code small— that is to say, the number of logically distinct data areas which it changes or depends on.

This helps in debugging by making you aware of the possible areas a piece of code can affect. It is often not easy to achieve, but use of procedures can help.

In general, FORTRAN offers two levels of names—those within a procedure, and those which are global-names of procedures and labeled common. Try not to overload either the global or local category. FORTRAN also offers some very positive gains when compared with Algol-like languages, and you should realize that a program is not necessarily better structured for being written in Algol or Pascal. Two main benefits of FORTRAN will be mentioned.

Firstly, if we have three distinct areas of data, A, B, and C, and three procedures P, Q, and R, and the structure of the problem is such that P must access A and B, Q must access B and C, and R must access A and C, then the FORTRAN program can very easily reflect this by making A, B and C separate labeled common blocks, and declaring the appropriate ones in each routine. In Algol-like languages, this is not possible, and A, B, and C are made global (and hence available) to all routines. This is a specific example of the general problem which causes so many large Algol programs to begin with an immense list of global variables, most of which are really local to just two or three routines. In your FORTRAN program you can avoid this difficulty, because the language is rich enough.

Secondly, in FORTRAN it is relatively easy to read a piece of code and to discover where various branches go to. In Algol-like languages it is almost impossible unless the writer has been scrupulous in 'correctly' indenting his program to show clearly where, for example, a particular ELSE clause terminates.

(Of course, determing in FORTRAN that the branches do, in fact, reflect an IF THEN ELSE structure is more difficult, unless comments or indentation are provided, but this important disadvantage does not remove the benefit of easily identifying the destination of a branch.)

This section cannot be completed without mention of the use of empty lines (with a C in column 1!), and of indenting spaces, to make a program more readable.

It is not easy to produce a program with a good structure, but it is well worth the effort for the saving in debugging time, and the ease of later modification.

7. DEBUGGING AND TESTING

In this section we mention some of the techniques available, and some system facilities often provided, to help with the debugging and testing of a program.

7.1 Program Debugging

There are four important system facilities which can help in debugging a program. If a system does not have them, there is little that can be done except to bring pressure to buy machines that do provide the facility. (If there are any budding compiler-writers reading this text, then they can do something about it by building the features into the system from the start.)

7.2 Expensive Run-Time Checks

There are a number of checks which can be applied at run time. The most useful by far is checking for array subscripts going out of range: this is a costly check, and it should be possible to run without it. Other checks of this nature are on the use of a variable before it has been set, and on the matching of type and number of arguments on a subroutine or function call.

One will sometimes find that an installation has a 'debugging compiler' which puts these checks in but whose compiled code runs far too slowly for real work. The 'optimizing compiler', on the other hand, may have no debugging aids at all. This makes errors which occur after, say fifteen minutes' machine time, very difficult to deal with. Some self-help in large programs is usually worthwhile; checking array bounds on every access would of course be tedious, but there are usually rather few sensitive areas in a program where integer variables to be used as subscripts are either input, calculated from data or incremented a number of times based on data. It is easy, and very worthwhile to put in checks to ensure the variable never goes beyond the size of the arrays it will be used for (e.g. checking on stack overflow and underflow).

7.3 Tracing

At its crudest, this is simply a facility for printing out a record of the transfers of control taking place in a program. To be useful three features are needed:

(1) Pre-dated switch on. By this we mean the ability to say 'Give me a trace starting 100 jumps before this current point in the program'. (In other words 'How did I get here?' rather than simply 'Where am I going?')

(2) The trace information should be in terms of the source program, not in terms of the compiled code, and should omit any transfers made within system routines, library routines, etc.

(3) The trace information should be pre-digested. By this we mean that cycles, however formed, should be recognized and a trace of, for example

```
DO  1  I = 1,3
DO  2  J = 1,3
```

```
2  ...
   DO  3  K = 1,4
3  ...
1  ...
```

should be printed out as

```
CYCLE 1  3 times
```

where CYCLE 1 is DO 1 to DO 2

```
CYCLE 2  3 times

LABEL 2  to DO 3

CYCLE 3  4 times

LABEL 3  to LABEL 1
```

where CYCLE 2 is DO 2 to LABEL 2
where CYCLE 3 is DO 3 to LABEL 3

[Machines existed in the early 1960's which gave information in this form (See [5]) but such diagnostic output is extremely rare even today!]

7.4 Checkpointing

This is a mechanism for obtaining printouts of certain variables at various points in the program. There are a number of variants of the facility in use.

(1) Requesting the system to print the values of specific variables whenever they change.

(2) Flagging certain assignment statements with keys which result in compiled code such that when the statement is obeyed, the value set in the variable will be printed (and the variable name) if the corresponding key is 'set'. The keys can be set and unset by the program. This tends to produce a large amount of undigested (and indigestible!) output.

(3) A key mechanism as in (2), but applied to WRITE (hopefully free-format) statements, such that the statement is only obeyed if the key is set. A further feature at compile time which (optionally) causes all keyed statements to be completely ignored makes this a very useful facility, although for transportable programs the statements would need to be deleted.

Once again, in the absence of a system facility, some self-help is possible.

It is a common trait among all programmers to think that they have a working program; if not first time, then at least long before they actually have. The net result is that people usually err on the side of producing too little output to assist debugging rather than too much.

Whenever any fairly complicated process has been used to produce an array, it is known roughly what the array should look like, and the program is going to continue with a later part based on this array, then it is foolhardy not to print out at least a summary of the array. A full printout at some stage in the debugging/testing process will often pick up bugs which would otherwise be completely undetected.

7.5 Postmortems

The final debugging aid–certainly the most valuable if properly done, and potentially the cheapest–is the postmortem of a program. Postmortem facilities again come in great variety, ranging from a crude and almost useless hexadecimal dump and machine oriented back–trace, to an elaborate and very helpful facility [6, 7]. We describe here a 'good' system, and the reader can see if his own measures up to this standard!

Two points deserve mention:

(1) In many machines, provided the system clears the store to binary zero, this will be a non–standard floating point number, and the system can assume, if any variable contains this, that it has never been written to. Its value can therefore be printed as 'unset'. Where this is not possible, an array which has zeros above some point can be considered as 'unused' from that point. We will talk therefore about the 'top' of an array—the last location which is either set, or non–zero, depending on the machine. This is often a useful figure to know.

(2) The simplest way to correlate the position in core at which an error (any error) occurs, with the position in the source, is to compile instructions setting some register or location to the current 'line–number'. Similarly, to print out the values of source variables at the end of the program demands the presence of a symbol table, giving the names of variables and where they are held. Although this is the simplest approach, if efficient use of core is a prime concern, details of both the symbol table and the mapping from core to source can be held on disk.

Let us now return to 'line–number'. It will usually be necessary to pack into one word both the last statement number set and the number of source lines (or statements) since that point. The latter value will have to be modulo some number–word lengths being finite. Suppose seven bits are available. Then two approaches are possible.

(a) Numbers 0 to 127 are printed as e.g.

```
'in line 36'.
```

Numbers over that are held modulo 128, i.e. as numbers 0 to 127 and therefore line 128 comes out falsely as

```
'in line 0'.
```

(b) The seventh bit is used to indicate

'beyond 64' and we have

Number 0 to 63 printed as, e.g.

'in line 36'.

Numbers 64 to 127 are printed as, e.g

'in line 36 (modulo 64)'.

Thus line 128 (and line 64) would come out as

'in line 0 (modulo 64)'.

We have labored this point, because it is again a good example of how a little thought can easily improve a user interface. Schemes (a) and (b) *both* distinguish a line completely if there are less than 128 lines between labels. But for more than 128, scheme (a) makes a wrong statement, scheme (b) actually tells the user it is 'modulo 64' and reminds him of information he probably read once and forgot. Both schemes use precisely the same number of bits.

We can now return to the postmortem facility. There must be a balance here between producing too much output (e.g. listing all elements of every array) and producing too little. A good compromise is to print all simple variables, and all arrays with less than, say, ten elements (five to a line say), and to summarize all other arrays. The summary should give the position and value of the following.

(i) The 'top' as defined earlier.
(ii) The maximum item (its position and value).
(iii) The minimum item (its position and value).

The printout should give the variables of the routine containing the error, then of the routine calling it, and so on, back to the main program. It only requires a small refinement to prevent information in COMMON (or arguments) from being printed more than once.

There are three main extensions to this feature. The first is to allow activation of the printing (for variables in the current routine only), during the running of the program. This is not very useful. The second extension is to allow the user to provide write statements (called POSTMORTEM (———,———) transfer list, say) which are activated as part of the post-mortem printing of any routine. The third extension demands an on-line system, and causes the program to be dumped, and allows the user to specify what he wants printed. It is often, in this case, possible to set a variable and restart the program at any label in any of the currently active routines. However, this is an expensive facility to provide, and of debatable value.

In this area there is much less scope for direct aid from the system, since the requirement here is to generate data such that the whole of the program is 'exercised', and to be able to check the results afterwards. There are two main techniques, and both should usually be applied. The first, which we call 'random testing', is to use a random number generator to produce options and data for the program. This approach can be seen as a soak test, and the amount of checking of results may have to be minimal. Often, however, a second, simpler, program can be written which will produce the input data from the results and this can be used to check the answers in the soak test. (See [8], pp. 63—70 for further discussion of this technique.)

The second technique, and perhaps the most important, we call 'complete testing' (although the word complete is slightly too strong, it seems the right one). In this phase we aim to supply data to exercise every single statement in the program—and further than that, to cause every branch instruction (IF) in the program to branch in every possible direction at least once. This is where mechanical assistance is needed to record which branches have been taken. The author has available a simple program written in SNOBOL which will take a FORTRAN program and insert tracing statements in each subroutine. The basic technique is to recognize branch statements (GOTO, IF, etc.) and to insert in the branch an instruction setting an element of an array (IT in the example below). At the end of the job the user simply prints out IT, and zero values indicate branches which have not been taken. More data, especially designed to provoke these branches, can then be thought out. With a complex program it can happen that, when thought is given to a particular branch, it is discovered that it can never be taken. The basic technique is illustrated by the following:

```
      ...
      ...
      IF (A) 1, 2, 3
      ...
      ...
      END

      SUBROUTINE JACK
      ...
      ...
      IF (A) 1, 2, 2
      ...
      ...
      END
```

would be turned into

```
      COMMON/IT/IT(1000)
      ...
      ...
      IF (A) 1001, 1002, 1003
      ...
      ...
1001  IT(1) = 1
      GOTO 1
```

```
1002  IT(2) = 1
      GOTO 2
1003  IT(3) = 1
      GOTO 3
      END

      SUBROUTINE JACK
      COMMON/IT/IT(1000)
       ...
       ...
      IF (A) 1001, 1002, 1002
       ...
       ...
1001  IT(4) = 1
      GOTO 1
1002  IT(5) = 1
      GOTO 2
      END
```

Again, it cannot be too strongly emphasized that it is no use exercising all parts of the program unless the answers produced are examined.

There is a strong similarity between the instructions needed to be inserted for the 'complete testing' and the instructions needed for identifying inner loops. Any system facility would usually incorporate both.

Finally, note that when a program is exported to another installation it should always be supplied with the data (and any testing programs you produced) for complete and for random testing, and also the results obtained.

8. IN CONCLUSION

This chapter has attempted to mention some of the points which a good FORTRAN programmer should be aware of when he is writing his packages. It is impossible to lay down firm DO's and DONT's in many of the areas concerned, and the author has not attempted a list of hard and fast rules. Rather, an attempt has been made to highlight the possibilities and problems so that the individual or organization can develop a good style of programming, conventions, and rules. An intelligent acceptance by a programmer of the need for clarity, good user interfaces, and a robust program is probably more useful than any number of rigid rules. This chapter has aimed to promote such an intelligence.

9. REFERENCES

[1] D. W. Barron. *Recursive Techniques in Programming.* American Elsevier Publishing Company, Inc., N.Y. (1968).

[2] J. R. Bitner and E. M. Reingold. "Backtrack programming techniques." *Comm ACM 18,* 11(Nov 1975), 651—656.

[3] J. Larmouth. Variable Precision Arithmetic in FORTRAN. Computer Laboratory, University of Cambridge, 1971.

[4] D. E. Knuth. *The Art of Computer Programming, Vol. 1.* Addison—Wesley, Reading, Mass., (1968).

[5] D. W. Barron and D. F. Hartley. "Techniques for program error diagnosis on EDSAC 2." *Computer J. 6,* (1963), 44—49.

[6] E. Satterthwaite. "Debugging tools for high level languages." *Software—Practice and Experience 2,* (1972), 197—217.

[7] P. Hazel, J. Larmouth and A. Stoneley. "Some comments on FORTRAN Systems." *Software—Practice and Experience 3,* (1973), 185—192.

[8] P. Naur. *Concise Survey of Computer Methods.* Mason & Lipscombe Pub. Inc., N.Y. (1974).

[9] E. Horowitz and S. Sahni. *Fundamentals of Computer Algorithms: Design and Analysis.* Computer Science Press, Potomac, Md. (1978).

A Glance at FORTRAN 77

Frances E. Holberton

The new FORTRAN 77 revised standard contains changes and additions to FORTRAN. Adapting older FORTRAN programs to conform to FORTRAN 77 requires that several items be kept in mind, particularly the incompatibility of Hollerith, changes in intrinsic functions, and the enhanced procedural features. A background in 1966 FORTRAN is assumed for discussions.

Key Words: ANSI; FORTRAN; FORTRAN extensions; Standard FORTRAN.

1. INTRODUCTION

The FORTRAN 77 document [6] describes the language as viewed by a user rather than as seen by an implementor of a processor. There is a distinction between a 'standard-conforming' FORTRAN 77 program and a 'standard-conforming' processor. If a user writes a program employing only the forms and relationships as described in [6], and the program adheres to prohibitions during execution, it is said to be 'standard-conforming.' However, the standard is permissive. A 'standard-conforming' processor may accept extensions for extra statement types, intrinsic functions, and condition definitions that FORTRAN 77 leaves undefined. Nonetheless, such extensions should not cause a 'standard-conforming' program to be interpreted erroneously. Even if a processor performs compile-time detections of non-FORTRAN 77 usage, such enforcement can miss other conditions such as a non-standard I/O format (developed during program execution), or a subscript out of range. A FORTRAN program using extensions or relying upon 'undefined' conditions is not 'standard-conforming'.

Although the primary purpose of the new standard is to promote portability of FORTRAN programs, there remain features that can inhibit portability. A standard-conforming program might not be acceptable to a different processor if it uses more than the FORTRAN character set or relies on a collating sequence, file name or I/O unit number, none of which are specified in the standard.

2. CONFLICTS BETWEEN FORTRAN 66 AND FORTRAN 77

A primary goal in developing FORTRAN 77 was to minimize conflicts with FORTRAN 66. Changes that have been introduced correct errors in the previous standard or strengthen FORTRAN in a significant manner. Two important areas where a FORTRAN program cannot satisfy both FORTRAN 66 and FORTRAN 77 are Hollerith data and intrinsic functions. The following listing of conflicts should alert FORTRAN users of potential conversion problems and assist in preparing new programs that will be adapted to FORTRAN 77 as compilers become available. Additional details on FORTRAN 77 features appear in Section 4.

2.1 Hollerith Data

Hollerith constants and Hollerith data *are not permitted* in FORTRAN 77. To improve program portability, machine dependent Hollerith has been replaced by an explicitly declared character type. It is expected that most implementors will retain Hollerith constants and data to support older programs. This change in the FORTRAN language can affect an old program in many ways; replacement may require more than a mechanical substitution. Understanding the significant differences between Hollerith and character data type can ease the transition to FORTRAN 77. Programs written in the interim using Hollerith can be easily adapted to CHARACTER for a FORTRAN 77 compiler, provided the following contraints are observed.

CHARACTER in FORTRAN 77 may not be equivalenced to non-character data. If older Hollerith data, under the guise of integer or real data type, uses the same data name for an integer or real datum, then the Hollerith occurrence must now be stored elsewhere. In addition, if any of the data in a common block is of type CHARACTER then all must be. This is also true for blank common; a separate common block must be designated to hold Hollerith data as characters.

A character constant is a character string enclosed in apostrophes, while a Hollerith constant in FORTRAN 66 has an exclusive form nH. This difference dictates that 'nH' forms of Hollerith constants in a DATA statement must be changed. An extension on many existing FORTRAN 66 compilers permits a quoted character string.

Hollerith data in relational expressions other than .EQ. and .NE. must be examined for possibly different results when converted to CHARACTER, even on the same machine, because Hollerith data is left justified in storage. On some word computers the high order bit of the left-most character may occupy a sign position: Hollerith data, being type integer or real (FORTRAN 66 has no type HOLLERITH), is treated as if it were a signed numeric causing the relationship to be numeric rather than character. The CHARACTER type in FORTRAN 77 provides for a proper interpretation of a character relational. FORTRAN 77 defines a partial collating sequence that is not specified in FORTRAN 66: character 'blank' (or space) precedes the number and alphabetic sets. The relationship between the number set and the alphabetic set is not defined, except that the two may not overlap. The order of the special characters is not treated. While this new requirement does not affect those systems that use EBCDIC or ASCII as internal codes, it may plague others that position 'blank' in the collating sequence such that it can not conform as prescribed. FORTRAN 77 does not state that 'blank' is the lowest ordered character in the processor character set, or that the result of a relational expression other than .EQ. or .NE. will be the same for different internal character representations. However, FORTRAN 77 does provide a set of logical intrinsic functions that perform a character relational with the ASCII collating sequence.

2.2 Intrinsic Functions

FORTRAN 66 has two sets of functions: Intrinsic (built-in) Functions and Basic External Functions. The names of the integer and real functions conform to the FORTRAN 66 implied type convention. However, while the 1966 standard has functions of type double precision and complex, there are no implied typing rules for these functions, despite double precision function

names beginning with the letter 'D' and complex with 'C'. The 1966 standard states that FORTRAN processors must supply the external functions listed in the Basic External Function Table, yet it is unclear whether type statements are required for double precision and complex functions. Not requiring a type statement could be interpreted as an extension, or alternately, the appearance of a type statement as redundant. According to the Basic FORTRAN specification [2], a function reference to CMPLX is classified as a real external function, while in the full language it is a complex intrinsic function. So FORTRAN 66 has an improper subset.

The Basic External Function class remained from the days when a user loaded the relocatable binary decks for external functions along with other external procedures; there were no double precision or complex types. The user who wanted a better function coded it in assembly language and loaded his version in place of the system's. When the functions were later placed on a library tape instead of loaded from cards, a different technique had to be provided for replacements. Because FORTRAN 66 sidesteps this problem, some implementors have changed the meaning of the 1966 standard: If a Basic External Function name appears in a type statement, this implies the user wants his own function. Other implementors load users' routines first; if a Basic External Function is not supplied, the system routine is used. This leads to a difficulty; for double precision and complex function names these types are applied to the expression analysis even when an explicit type statement does not appear, making these function names reserved names while the integer and real function names are not. FORTRAN 77 changes this.

FORTRAN 77 introduces mixed mode expressions, IMPLICIT statements, and generic names for intrinsic functions, so that FORTRAN functions names are now merged into a single set divorced from all naming conventions and type statements. The type of each intrinsic function is specified in the standard. FORTRAN naming conventions, IMPLICIT statements. and type statements apply to variables, arrays, statement functions and user external functions, but *not* to intrinsic functions.

In FORTRAN 66, if a Basic External Function or a user supplied external function is passed as an actual argument to an external procedure, that name must appear in an EXTERNAL statement that distinguishes the name from a variable. However, an intrinsic function passed as an actual argument in FORTRAN 77 must appear in an INTRINSIC statement. EXTERNAL is appropriate in FORTRAN 77 whenever an external procedure is user supplied. Thus, the FORTRAN 77 EXTERNAL statement designates a user-supplied routine both for function references and actual arguments; this removes the function name from the reserved class and draws a distinction between user functions and system supplied functions. This division permits implementors to use special hardware, calling sequences or optimization. Because names of intrinsic functions and implementor extensions to the intrinsic function table take precedence over a user supplied function name, the EXTERNAL statement should be used to supply names of the user's external functions. This ensures uniform behavior when a program is transported to a different environment.

2.3 DO-Loops

The concept described in FORTRAN 66 as the 'extended range of a DO' has been deleted from FORTRAN 77. In FORTRAN 77, the range of a DO-loop consists of all of the executable statements that appear following the DO statement that specifies the DO-loop, up to

and including the terminal statement of the DO-loop and precludes a transfer of control into the range of a DO-loop from outside. The range of a DO-loop may be entered solely upon execution of a DO statement. The extended range of a DO is permitted in FORTRAN 66 only with a specific kind of nesting of DO statements called a 'completely nested nest:' The results are non-structured programs. Sequences of statements to handle special conditions for which extended ranges were often used must now appear within the range of a DO. For the rare case in FORTRAN 66 when an extended range of a DO is used to contain a closed internal procedure executed both from the DO-loop and external to the DO-loop, the procedure must now be expressed as an external procedure. Invocation of an external function or a CALL statement from within a DO-loop is not considered an extension of the range.

With FORTRAN 66, if a terminal parameter K of a DO statement is less than the initial parameter J as in

```
DO 10 I = J, K
```

then the program is not standard-conforming. Many processors permit the above as an extension; the DO-loop executes once. During development of FORTRAN 77 the international FORTRAN community was surveyed on whether the DO-loop in the above case should be performed once or not at all. The response divided almost evenly with a slight preference for zero times, which was adopted. Therefore in FORTRAN 77, whenever a terminal parameter is less than its corresponding initial parameter, the DO-loop is skipped. To execute a DO-loop at least once and be acceptable to both FORTRAN 66 and FORTRAN 77 the following sequence of statements is necessary:

```
K1 = MAX0(J, K)
DO 10 I = J, K1
```

while in FORTRAN 77, which permits both generic functions and expressions as DO-parameters, the following is sufficient:

```
DO 10 I = J, MAX(J, K)
```

2.4 Subscript Expressions

In contrast to FORTRAN 66, FORTRAN 77 demands that the value of each array subscript expression not exceed its corresponding upper bound declared for the array name in the program unit. FORTRAN 66 does not specify any restriction on the maximum value of a subscript expression, except that a subscript must not exceed the maximum length of an array. A published clarification for FORTRAN 66 permits the following:

```
      REAL A(10,5)
      DO 1 J = 1, 50
1     A(J, 1) = 0.0
```

while in FORTRAN 77 the example should be changed as:

```
      REAL A(10, 5), B(50)
      EQUIVALENCE (A(1,1), B(1))
      DO 1 J = 1, 50
1     B(J) = 0.0
```

A compiler is not required to detect an improper reference, but compilers with bounds-check capability can now perform this service.

2.5 EQUIVALENCE Statement

FORTRAN 77 allows only those arrays declared as one-dimensional to have one-dimensional subscripts in an EQUIVALENCE statement. For example:

```
REAL A(2,3,4), B(4,8)
EQUIVALENCE (A(22), B(1,1))
```

A(22) is *not* allowed in FORTRAN 77, but will work in FORTRAN 66. Some early FORTRAN compilers made users expand subscript expressions and record the result as a one-dimensional subscript in an EQUIVALENCE statement. This form is rarely used today; it was removed from FORTRAN 77 because it is error prone.

2.6 Input/Output

Reading into a format specification

The capability to read directly into an nH edit descriptor in a FORMAT statement has been removed from FORTRAN. FORTRAN 66 supports :

```
      READ (5,7)
7     FORMAT (14H FILL ON INPUT)
      WRITE (6,7)
```

The first fourteen characters from the record on unit 5 store directly into the format, and are then written onto unit 6. This facility is very costly and little used. There is no means for the altered format to be preserved if it appears in a subprogram, because in such a form it cannot be stored in a common block. FORTRAN 77 has format specifications contained directly in I/O statements; the example below is equivalent to the preceding, with an added facility, which in a subprogram, saves the character variable:

```
CHARACTER B*14
SAVE B
READ (5, '(A)')B
WRITE (5, '(A)')B
```

where '(A)' is equivalent to the FORMAT statement: FORMAT (A14)

Records in a sequential file

In contrast to FORTRAN 66, the new FORTRAN 77 forbids sequential files that contain both formatted and unformatted records. To require a system to support mixed records is not considered necesary for program portability; to do so would be a step in the wrong direction for portable data.

Redundant parentheses in an I/O list

FORTRAN 77 prohibits a simple I/O list enclosed in parentheses from appearing in an I/O list. This is permitted in FORTRAN 66. FORTRAN 77 allows expressions to appear in output lists, but requires that parentheses enclosing more than one I/O list item must mark an implied DO-loop. This restriction is imposed to eliminate potential syntactic ambiguities introduced by complex constants in list-directed output lists. As all the parentheses referred to are redundant, a FORTRAN 66 statment converts to FORTRAN 77 upon deleting redundant parentheses enclosing more than one item in an I/O list.

Definition of entities in an input list

Consider the following example:

```
REAL A(25)
EQUIVALENCE (I, J)
J = 10
READ (5) I, A(J)
```

FORTRAN 66 interprets array element A(J) as A(10), whereas FORTRAN 77 reads the input value into A(I), with I's value just determined. This is because FORTRAN 66 delays the definition of an associated entity (equivalenced, such as J above) until the end of execution of the input list. J is not redefined until the completion of the READ statement even though I, which is also associated, is defined during reading. Thus the statement from the preceding example:

```
READ(5)I,A(J)
```

which is different from

```
READ(5)J,A(J)
```

or

```
READ(5)I,A(I)
```

in FORTRAN 66, is interpreted to be the same in FORTRAN 77. In FORTRAN 77, the definition of an associated entity with an entity in the input list occurs at the same time as the definition of the list entity. The concept of 'second level definition' that applies to subscripts and computed GOTOs in FORTRAN 66 has been discarded.

Negative valued I/O unit identifier

In some implementations a special meaning is applied to a negative valued unit identifier. FORTRAN 66 does not explicity prohibit negative valued unit identifiers, but FORTRAN 77 *does.*

Negative signed zero on edited output

One published interpretation of FORTRAN 66 specifies that if the internal representation of a real or double precision value is negative, the external representation of this value, even if truncated to all zeroes, must contain a negative sign. This is changed so that FORTRAN 77 produces edited data acceptable to the American National Standard for Numeric Values in Character Strings for Information Interchange (X3.42-1975). A negative signed zero is never produced on edited output. Also, to support X3.42-1975, an E exponent on output must be explicitly signed. A space character is not used in lieu of an explicit plus sign.

Writing after an endfile record

FORTRAN 66 does not expressly prohibit writing a record after an endfile record; however, no interpretation is given for reading an endfile. FORTRAN 77 forbids the writing of a record after an endfile record, but does permit reading of an endfile record, and provides an interpretation for reading of an endfile record. The new Standard [6] has further details on this.

2.7 END Statement

An END statement may not be labeled in FORTRAN 66; therefore, the labeled statement:

```
100  END
    -FILE 6
```

fails as an END statement. However, in FORTRAN 77, the END statement may be labeled; consequently, the first line in the example qualifies as an END statement and not as an ENDFILE 6 statement.

2.8 BLOCK DATA Subprogram

FORTRAN 77 permits a BLOCK DATA subprogram to be named. It does not allow more than one unnamed BLOCK DATA subprogram to appear in an executable program. In contrast, FORTRAN 66, which does not contain this prohibition, can be interpreted to permit more than one BLOCK DATA subprogram.

2.9 Blank Lines

An interpretation of FORTRAN 66 specifies that blank characters in columns 1 through 72 constitute an initial line of a statement that must be followed by a continuation line. By public demand this interpretation is abandoned in FORTRAN 77. A completely blank line is a comment.

2.10 Columns 1-5 of a Continuation Line

Another published interpretation of FORTRAN 66 specifies that columns 1 through 5 of a continuation line may contain any character from the FORTRAN character set, except that column 1 must not contain the character 'C'. Some implementations permit this; it is often used to number continuations, but it is dropped from FORTRAN 77, so that a continuation line must have blanks in columns 1 though 5. It is felt that retaining numbered comments would impede any future revision to a free-form FORTRAN text.

3.0 PORTABLE NON-STANDARD PROGRAMS

Undoubtedly many more portable FORTRAN programs exist today than actually conform to the FORTRAN 66 standard. The most prevalent discrepancy between a standard-conforming FORTRAN 66 program and a typical portable program lies in the relationship between the main program unit and its subprograms. While programs that count on the retention of storage for a subprogram may be portable over a wide class of computers, they are not standard-conforming. Revising such programs to meet the standard requires introduction of a labeled common block to hold local variables whose values must be preserved deliberately. The initialization of data must be moved to a BLOCK DATA subprogram, and the names of all labeled common blocks must appear in a COMMON statement in the main program. Because this kind of revision is quite time consuming, the modifications are not often undertaken unless a program is to be executed on a FORTRAN system that overlays external procedures without updated storage.

3.1 Storage

The FORTRAN 66 standard does not typify common practice of its period in the treatment of storage in a subprogram. Earliest systems ran only a single program at any one time; a complete program was resident within the computer unless a user-directed overlay procedure was invoked. However, to reconcile FORTRAN 66 to systems with small memories and disk secondary storage, and to accommodate multiprocessing environments, the subprogram and its data storage stand as subordinates to the main program. Automatic overlay of subprograms can occur without preservation of data values. This decision has caused a serious break with earlier systems, dividing the FORTRAN user community into two divergent groups.

A significant user investment in programs and market competition forces most implementors to continue the retention of a complete program in memory, with no subsequent relocation of subprograms during the program execution. Thus, local variables defined in a subprogram remain valid on subsequent use of the subprogram. FORTRAN 66 specifies the DATA statement as the 'DATA initialization statement' permitting a variable to be given an initial value that can be later changed, but only in a main program does the Standard fully define the value if changed. (Refer to Chapter 2, Section 5.0 for details on definition and undefinition of values in FORTRAN 66.)

Another frequent nonstandard feature found in otherwise conforming programs is an unsubscripted array name in a DATA statement, together with a list of values for the entire

array. FORTRAN 66 requires that each element of the array be specifically written. Most contemporary FORTRAN compilers permit the shortened form of the array name to represent all elements of the array in the DATA statement. The substitution of each array element name, although tedious, is a simple task. FORTRAN 77 permits unscripted array names in a DATA statement. The implied order of assignment is by column.

4.0 MAJOR EXTENSIONS IN FORTRAN 77

4.1 Program Structure

The following statements have been added to the language:

```
IF (e) THEN
ELSE IF (e) THEN
ELSE
END IF
```

The block IF (e) THEN statement is used with a corresponding END IF statement and, optionally, the ELSE IF (e) THEN and ELSE statements to control the execution sequence. The groups of statements delimited by the IF-THEN and the END IF must be properly nested, both with respect to other such groups and with respect to DO loops. Transfer of control is not permitted into such groups.

Computed GOTO default

If the value of the control expression of a computed GO TO is out of range, execution continues with the statement following the computed GO TO.

DO-loop parameters

The DO-variable may be an integer, real or double precision variable and similarly, DO parameters may be integer, real or double precision expressions. If necessary, values of parameter expressions are converted to the type of the DO-variable. The DO-variable first takes the value of the initial parameter, m1. The iteration count is:

```
MAX (INT ((m2 - m1 + m3 )/ m3 ), 0)
```

where m2 is the terminal parameter and m3 , the increment. The DO-loop will not be performed whenever

```
m1  >  m2  and  m3  > 0   or
m1  <  m2  and  m3  < 0
```

Upon completion of a DO-loop, the DO-variable is defined as incremented by m3 and available for use. m1 and m2 may be positive, negative or zero. m3 may be positive or negative, but not zero. A transfer of control is not permitted into the range of the DO.

Comment line

An asterisk or 'C' in column 1 or a totally blank line designates a comment line.

4.2. Procedure Names, Data Names, and Character Set

The apostrophe and the colon are included in the FORTRAN character set. The collating sequence is partially specified.

The main program may be named by a PROGRAM statement.

A block data subprogram may be named, and at most one unnamed block data subprogram may be included in an executable program.

The IMPLICIT statement may be used to declare implicit types for variables, arrays, statement functions, user defined functions and names of constants (PARAMETER statement), as specified by the initial letter of name.

The EXTERNAL statement may be used to both pass user-defined functions and subroutines to an external procedure and declare a name to be a user-defined external procedure, thus overriding intrinsic functions and processor additions to the FORTRAN-specified intrinsic functions. Automatic typing of intrinsic functions causes intrinsic function names in a function reference to take precedence over a user-defined function of the same name, unless that name appears in an EXTERNAL statement. Thus, the user can control the names of his external procedures without knowledge of the environment of execution or the processor extended library of functions.

The INTRINSIC statement, a specification statement, is used when a FORTRAN intrinsic function is passed as an argument to an external procedure. INTRINSIC may be used to denote FORTRAN intrinsic functions or additional processor supplied functions.

4.3 Data and Environment Control

Character data type

FORTRAN 77 includes character strings of fixed declared length. This permits variables, constants, arrays, and functions to be declared as type CHARACTER. The sole string operator is concatenation. There are intrinsic functions for conversion between single characters and small integers, for pattern matching (INDEX) and determining the length of a string (LEN). Four intrinsic functions of type logical, which compare two character expressions according to the ASCII collating sequence, provide character manipulation on a uniform basis regardless of any internal collating sequence.

Expressions

Arithmetic expressions may include subexpressions of mixed types. The data type of the result of a subexpression of mixed type is determined by rules of order, from lowest to highest as: integer, real, double precision, and complex. Consider a subexpression containing both integer and complex type. Conversion from integer I to complex type can be expressed as CMPLX(REAL(I),0.0) where CMPLX is the intrinsic function for forming a complex quantity from reals. REAL is a generic function that is the same as FLOAT for an argument of type integer. Conversion takes place before the subexpression is evaluated. A subscript expression is any integer expression. While A(1.E1) is not permitted because the subscript is a real expression, A(INT(1.E1)) is valid.

Arrays

An array has up to seven dimensions, with each dimension subscript bounded (lower and upper) via integer expressions. If only one bound is present, a default of one is assumed for the lower bound. The upper bound of the last or only dimension of a dummy (formal) argument may be left unspecified (via an asterisk).

DATA statement

Names in a DATA statement may include variables, arrays, array elements, character substrings, and implied DO-lists. An arithmetic constant is converted to the type of its associated data name by rules of arithmetic assignment. However, if a data name is double precision and its constant is real (does not contain the D exponent form), the processor may (optionally) supply more precision from the constant than real implies. If a program must be altered to improve the precision by changing the data names from the type real to double precision, the form of the constant in the DATA statment need not be changed and any excess precision expressed by the real constant form may be applied to the double precision value. While a program written in the full language can always prepare for the possibility of such a type change by using the D exponent form in the DATA statment, this form is not available in subset FORTRAN 77, which has no double precision type. Thus, a program written in subset FORTRAN 77 and transported to a full language processor with limited precision for real may be readily adapted to double precision without changing the form of the constants.

Compile time constants

The PARAMETER specification statement is used to give a constant a symbolic name. Constants might need change from one compilation to another, yet they are invariant during execution. A symbolic parameter may be used in some specifications, such as in the expression of an array declaration, or the length in a character type statement or IMPLICIT statement for character type, or subsequent PARAMETER statements: It is also useable in DATA statements, as well as any executable statement where a constant or an expression is permitted.

Data under format control

FORTRAN 77 introduces a number of features that ease data preparation and output editing. In FORTRAN 66 formated input, all trailing blanks are interpreted as zeros; all values must be right-justified in their fields. FORTRAN 77 provides for reading blanks in numeric data as null; this permits data to be left-justified in fields, and reduces some tedium of data preparation.

Input data may contain more precision than a processor can provide in corresponding real or double precision storage. The edit descriptors F, E, D, and G apply to real and double precision values. The user may control printing of digits and sign in an exponent part of the value via an Ew.dEe format. Explicit and relative tabbing and control termination of the format rescan shape output fields.

Generic functions

FORTRAN 77 provides for using the same name of an intrinsic function, its generic name, with arguments of different types. Thus, the generic function name ABS may be used with integer, real, double precision, and complex arguments, thereby supplanting IABS, ABS, DABS, and CABS. The result takes the same type as the argument. Divorcing intrinsic functions from FORTRAN naming conventions permits such generic names as MAX, MIN, LOG, and LOG10 to handle more than one type of argument. The type conversion functions INT, REAL, DBLE, and CMPLX may be used with any arithmetic type of argument to produce direct conversions. These conversions are the same as across the equals in the assigment statement. Generic functions are easier to remember and, again, they simplify program transportability whenever increased precision of real values must be provided under double precision type.

4.4 File Structure and Control

FORTRAN 77 has two kinds of files: external and internal. An internal file is accessed sequentially; it provides a means of transferring and converting data from one internal representation to another. A FORTRAN 66 extension that uses ENCODE and DECODE statements is similar, but the process is now performed via WRITE and READ.

There are two methods of accessing records of an external file: sequential and direct. The OPEN statement, which makes use of keywords to indicate specific properties, may be used to describe a file connection to a logical unit. For example,

```
OPEN(UNIT=10,ACCESS='DIRECT',FORM='FORMATTED',RECL=132)
```

describes a file on unit 10 as direct access, formatted with fixed length records of 132 characters. A CLOSE statement terminates the unit connection of a particular file. INQUIRE returns properties of a particular file or of the connection to a particular unit. If the inquiry is by *unit*, the following might be asked:

Does unit 10 exist?
Is there a file on unit 10?
Does the file have a name (Give me its name.)?
What is its access method?
Is it formatted or unformatted?

Similarly, an inquiry by *file name* might concern:

Does a file named 'ABC' exist?
Is it connected to a unit?
What is its unit number?
What is its access method?
If the file has not been connected to a unit
can it be connected for sequential access?
For direct access?
If it has been connnected for direct access what is
its record length?

A file connected for *direct access* can be queried for the next record number, a common request.

A PRINT statement is available. Local standard units for input and output are designated with asterisks, as in READ(*,123)A. There is list directed input and output, so that data can be converted on input and output without an explicit format specification.

4.5 Subprogram Interface

The current values of local variables, arrays, and entities in labeled common blocks are retained between invocations of a subprogram by appearance of their names in a SAVE statment in the subprogram. SAVE with no list preserves all subprogram data with their most recently defined values regardless of the method of implementation and system storage management.

The ENTRY statement is permitted in an external function or subroutine, and there is an alternate return statement, RETURN i, for subroutines.

An adjustable dimension may be passed to a subprogram through COMMON, and the last dimension of an array may be an assumed length in a dummy array declarator. In usual FORTRAN 66 practice such an array , A, might have been declared as A(1) or A(10000) depending upon the compiler— in FORTRAN 77 A(*) is sufficient.

Character functions and dummy arguments of type character may have their string length passed implicitly to the subprogram. Such a length may be different for each reference to the subprogram, and be available for use via the intrinsic function, LEN. No dynamic storage allocation is assumed by any of the character features in FORTRAN 77.

5.0 SUMMARY

The FORTRAN 77 language, although much larger than FORTRAN 66, is comparable to what many users expect on large computing systems. Conflicts between FORTRAN 66 and FORTRAN 77 are limited to those necessary to correct errors in the previous standard or to enhance the language significantly. The new extensions, while modest in scope, strengthen FORTRAN's utility for program development, maintenance, and portability.

6.0 REFERENCES

[1] *FORTRAN X3.9–1966.* American National Standards Institute, New York, (1966).

[2] *Basic FORTRAN X3.10–1966.* American National Standards Institute, New York, (1966).

[3] "Clarification of FORTRAN Standards—Initial Progress." *Comm. ACM. 12,* 5(May 1969), 289–294.

[4] "Clarification of FORTRAN Standards—Second Report." *Comm. ACM 14,* 10(October, 1971), 628–642.

[5] "Draft Proposed ANS FORTRAN." *ACM. SIGPLAN Notices 11,* 3(March 1976).

[6] *ANS FORTRAN X3.9–1978.* American National Standards Institute, New York, (1978).

[7] "FORTRAN vs. Basic FORTRAN." *Comm. ACM 7,* 10(October 1964), 591–625.

[8] Brainerd, W., "FORTRAN 77," *Comm. ACM 21,* 10(October 1978), 806–820.

☆ U.S. GOVERNMENT PRINTING OFFICE : 1980 O-311-046 (25)

NBS TECHNICAL PUBLICATIONS

PERIODICALS

JOURNAL OF RESEARCH—The Journal of Research of the National Bureau of Standards reports NBS research and development in those disciplines of the physical and engineering sciences in which the Bureau is active. These include physics, chemistry, engineering, mathematics, and computer sciences. Papers cover a broad range of subjects, with major emphasis on measurement methodology and the basic technology underlying standardization. Also included from time to time are survey articles on topics closely related to the Bureau's technical and scientific programs. As a special service to subscribers each issue contains complete citations to all recent Bureau publications in both NBS and non-NBS media. Issued six times a year. Annual subscription: domestic $17; foreign $21.25. Single copy, $3 domestic; $3.75 foreign.

NOTE: The Journal was formerly published in two sections: Section A "Physics and Chemistry" and Section B "Mathematical Sciences."

DIMENSIONS/NBS—This monthly magazine is published to inform scientists, engineers, business and industry leaders, teachers, students, and consumers of the latest advances in science and technology, with primary emphasis on work at NBS. The magazine highlights and reviews such issues as energy research, fire protection, building technology, metric conversion, pollution abatement, health and safety, and consumer product performance. In addition, it reports the results of Bureau programs in measurement standards and techniques, properties of matter and materials, engineering standards and services, instrumentation, and automatic data processing. Annual subscription: domestic $11; foreign $13.75.

NONPERIODICALS

Monographs—Major contributions to the technical literature on various subjects related to the Bureau's scientific and technical activities.

Handbooks—Recommended codes of engineering and industrial practice (including safety codes) developed in cooperation with interested industries, professional organizations, and regulatory bodies.

Special Publications—Include proceedings of conferences sponsored by NBS, NBS annual reports, and other special publications appropriate to this grouping such as wall charts, pocket cards, and bibliographies.

Applied Mathematics Series—Mathematical tables, manuals, and studies of special interest to physicists, engineers, chemists, biologists, mathematicians, computer programmers, and others engaged in scientific and technical work.

National Standard Reference Data Series—Provides quantitative data on the physical and chemical properties of materials, compiled from the world's literature and critically evaluated. Developed under a worldwide program coordinated by NBS under the authority of the National Standard Data Act (Public Law 90-396).

NOTE: The principal publication outlet for the foregoing data is the Journal of Physical and Chemical Reference Data (JPCRD) published quarterly for NBS by the American Chemical Society (ACS) and the American Institute of Physics (AIP). Subscriptions, reprints, and supplements available from ACS, 1155 Sixteenth St., NW, Washington, DC 20056.

Building Science Series—Disseminates technical information developed at the Bureau on building materials, components, systems, and whole structures. The series presents research results, test methods, and performance criteria related to the structural and environmental functions and the durability and safety characteristics of building elements and systems.

Technical Notes—Studies or reports which are complete in themselves but restrictive in their treatment of a subject. Analogous to monographs but not so comprehensive in scope or definitive in treatment of the subject area. Often serve as a vehicle for final reports of work performed at NBS under the sponsorship of other government agencies.

Voluntary Product Standards—Developed under procedures published by the Department of Commerce in Part 10, Title 15, of the Code of Federal Regulations. The standards establish nationally recognized requirements for products, and provide all concerned interests with a basis for common understanding of the characteristics of the products. NBS administers this program as a supplement to the activities of the private sector standardizing organizations.

Consumer Information Series—Practical information, based on NBS research and experience, covering areas of interest to the consumer. Easily understandable language and illustrations provide useful background knowledge for shopping in today's technological marketplace.

Order the **above** *NBS publications from: Superintendent of Documents, Government Printing Office, Washington, DC 20402.*

Order the **following** *NBS publications—FIPS and NBSIR's—from the National Technical Information Services, Springfield, VA 22161.*

Federal Information Processing Standards Publications (FIPS PUB)—Publications in this series collectively constitute the Federal Information Processing Standards Register. The Register serves as the official source of information in the Federal Government regarding standards issued by NBS pursuant to the Federal Property and Administrative Services Act of 1949 as amended, Public Law 89-306 (79 Stat. 1127), and as implemented by Executive Order 11717 (38 FR 12315, dated May 11, 1973) and Part 6 of Title 15 CFR (Code of Federal Regulations).

NBS Interagency Reports (NBSIR)—A special series of interim or final reports on work performed by NBS for outside sponsors (both government and non-government). In general, initial distribution is handled by the sponsor; public distribution is by the National Technical Information Services, Springfield, VA 22161, in paper copy or microfiche form.

BIBLIOGRAPHIC SUBSCRIPTION SERVICES

The following current-awareness and literature-survey bibliographies are issued periodically by the Bureau:

Cryogenic Data Center Current Awareness Service. A literature survey issued biweekly. Annual subscription: domestic $25; foreign $30.

Liquefied Natural Gas. A literature survey issued quarterly. Annual subscription: $20.

Superconducting Devices and Materials. A literature survey issued quarterly. Annual subscription: $30. Please send subscription orders and remittances for the preceding bibliographic services to the National Bureau of Standards, Cryogenic Data Center (736) Boulder, CO 80303.